PERFORMANCE EVALUATIONS FOR MULTICORE PROCESSORS

By

JULIUS JONGGARA R. HOT MARISI MARPAUNG

Bachelor of Science in Electrical and Computer Engineering
Oklahoma State University
Stillwater, OK
2003

Master of Science in Electrical and Computer Engineering
Oklahoma State University
Stillwater, OK
2006

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
May, 2012

PERFORMANCE EVALUATIONS FOR MULTICORE

PROCESSORS

Dissertation Approved:

Dr. Louis G. Johnson

Dissertation Adviser

Dr. R. G. Ramakumar

Dr. George Scheets

Dr. Blayne Mayfield

Outside Committee Member

Dr. Sheryl A. Tucker

Dean of the Graduate College

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

A significant amount of computer architecture research focuses on reducing the performance gap between processors and memories. A large main memory cannot provide instructions and data as fast as the clock speed of the processors. This has led to several innovative ideas to reduce the access time of instructions and data from the main memory to processors.

Cache is a temporary storage space used to fetch instructions and data from a main memory which is accessed frequently by processors in order to minimize the access time. As chip manufacturing technology improves, more transistors can be placed on a single chip. This enables hardware designers to place more processors and a hierarchy of bigger caches on a single chip while sharing a common external main memory. Several strategies for cache hierarchies have been proposed for these multi-core chips. A multi-level cache memory hierarchy with one of the levels shared by all the processors is widely used in commercial processors because it is the simplest way for processors to access the shared external memory.

A multi-core chip is typically defined as a system that has more than one processor. Sharing level 1, L1, caches is difficult because the L1 response time must be fast enough to keep up with the processors' clock speed. Sharing level 2, L2, caches among processors is more desirable because it enables processors to communicate with each other in a fairly short amount of time but without slowing the processors' clock speed. A multi-core chip with a single-shared L2 cache is the bsic configuration that will be studied in this dissertation. Other configurations will also be studied to determine whether better performance might be possible.

The goal of my dissertation is to use and improve a new simulation tool, Abakus, to study different cache hierarchies and configurations. Abakus can be used to evaluate the performance of any chosen processor and cache configurations. A significant part of this dissertation is devoted to validating the existing multi-core chips models that have already been developed within Abakus.

This dissertation is divided into 6 chapters. Chapter 2 discusses the literature review. Chapter 3 discusses the validation of our simulation tool, Abakus. Chapter 4 discusses the performance of shared bus multicore processors with several different cache memory configurations. Chapter 5 discusses the future work. Chapter 6 concludes the dissertation.

CHAPTER II

LITERATURE REVIEW

2.1 Uniprocessor's Cache Model

Figure 1 illustrates the connection between a processor with a L1 and L2 cache, and a main memory. Figure 2 shows the meaning of the cache symbols that will be used in cache memory system connection diagrams.

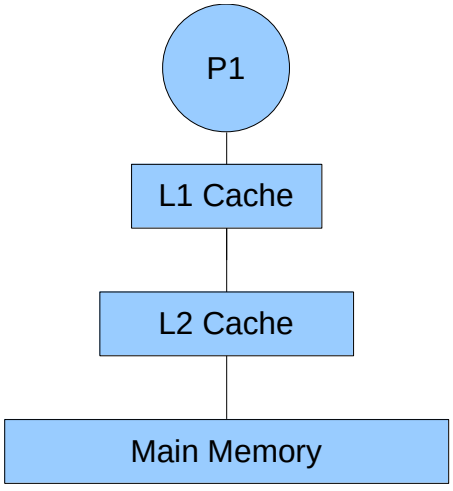

Figure 1. Uniprocessor Model

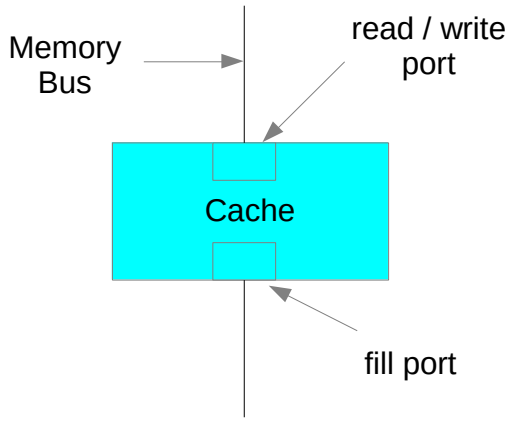

Figure 2. An example of a cache

In general, a cache has a read/write port to communicate with a lower level cache/processor and a fill port to communicate with a higher level cache or main memory. In a uniprocessor model, a processor will not share its data or instructions with any other processors; hence if the data or the instructions are not available in L1 cache, they can be fetched from the L2 cache or the main memory. Some exotic L1 cache models have been proposed to improve the performance of L1 cache [1 - 3] and they are out of the scope of this dissertation.

2.2 Multiprocessor's Cache Models

This subsection discusses various design alternatives for multiprocessor caches. These design alternatives have been used and are currently used in a multiprocessor system. As the number of processor that can be placed on a single chip increases, the cache architecture must be able to accommodate the processors by increasing the instruction and data bandwidth between the processors and the memory system.

2.2.1 Multiport Approach

In a multiport approach, a cache can have multiple ports to allow simultaneous read/write from upper level caches/processors. Figure 3 shows an example of a multiport cache.

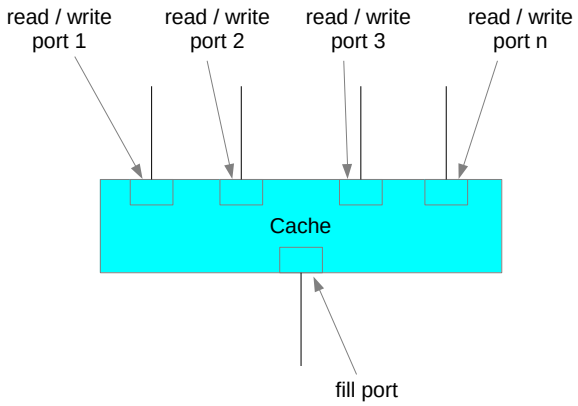

Figure 3. Multiport cache

Figure 4 shows the connection between four processors with their private L1 cache and a shared L2 cache.

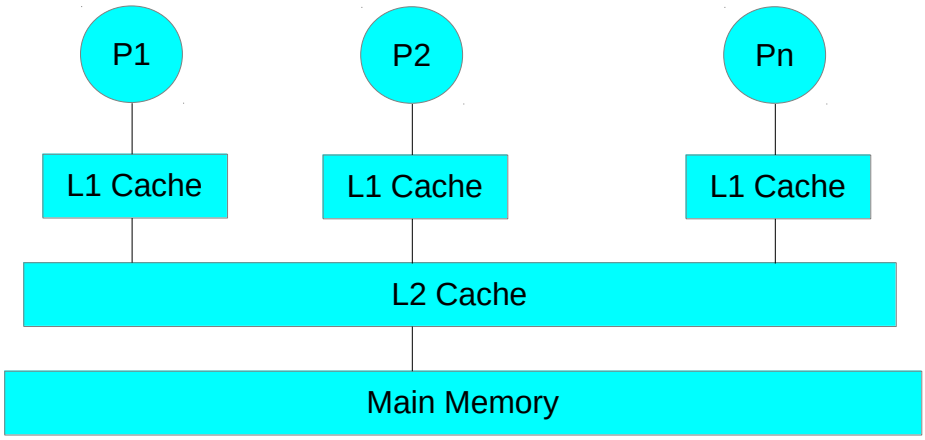

Figure 4. Multiport Architecture

In a uniprocessor model, one processor only needs one port/data or instruction bus line to access the data or instruction; hence you can only access one address at a time. In a multiport model, we want the processors to have the ability to access multiple numbers of ports in order to be able to read and write to different addresses at the same time. The problem comes when two or more processors try to access the same address. If one processor tries to write to it while the other one tries to read from it, which one should go first? Often time, designers use a directory-based coherence protocol to handle this issue. Snooping cannot be used in a multiport environment because there is not a single shared bus that can be snooped. A full map directory-based protocol is a cache coherence protocol that uses a presence vector, which is a vector of bits denoting where the cached copies reside [4]. This method reduces the time it takes to find a valid copy of data needed by a processor from other processors. The downsides of using this method are the amount of overhead storages needed as the number of processor goes up and the possibility of being a bottleneck as processors need to access it regularly. The full-map directory scheme of Maa et al [4] is presented in Figure 5. Two main disadvantages of using multiport memory approach are increase in the size of the chip with the number of ports and the significant power it consumes [5].

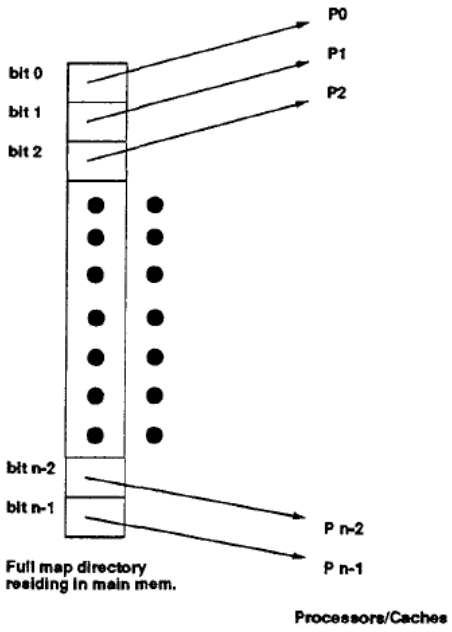

Figure 5. Full-map directory scheme [4]

Mattausch et al [5] also present a graph of the chip size vs the number of memory ports implemented in Hitachi Hokkai Semiconductor CMOS 0.5 μm process technologies in logarithmic scale as shown in Figure 6 that shows the multiport memory cell approach suffers from a scalability issue.

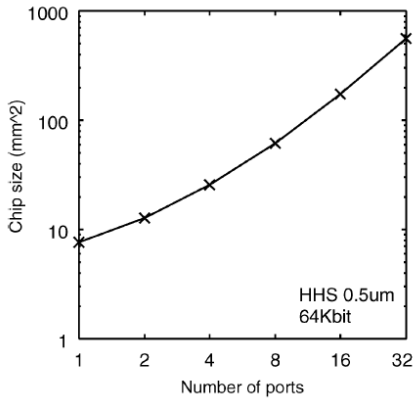

Figure 6. Ports vs chip size for multiport memory cell approach [5]

Researchers have spent an enormous effort in perfecting the multiport method [6-8]. One conflict that keeps arising is when multiple processors try to access the same address at the same time. Coming up with a way to fix the problem has been proven to be complicated and requires extra hardware. Cache designers then come up with an idea of using “banks” to remedy the problem.

2.2.2 Multibank Approach

From the outside, a multibank cache looks like a multiport cache. Each bank contains a unique address subset/partition of the whole L2 cache. Figure 7 shows an example of a multibank cache.

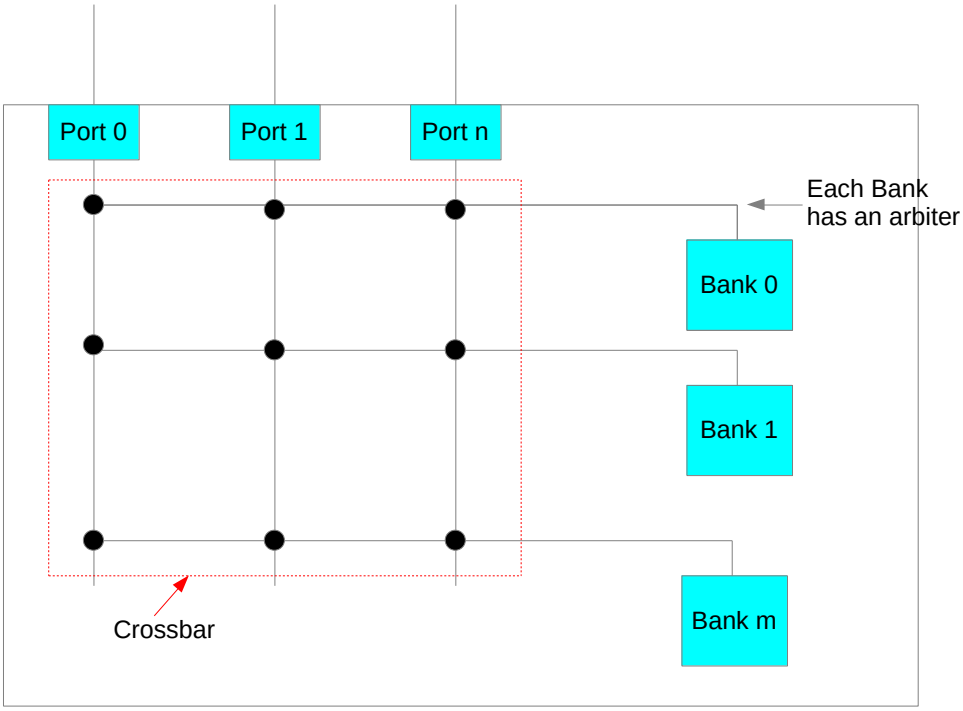

Figure 7. Multibank cache

The multibank approach also allows each processor to write and read to different ports at the same time, hence it is a relatively cheap and practical way to implement a multiport cache. The problem arises when two or more ports try to access one bank simultaneously and it is called bank conflict. It is possible to increase the number of banks to reduce bank conflicts but doing so will increase the chip size [5]. Recent research in multibank focuses on bank conflict avoidance and bank conflict resolution using scheduling, bank predictors, and queuing technique [9-18] and they are out of the scope of this dissertation.

Researchers and chip manufacturers often time use the idea of crossbar, and ring topology for connection among processors and caches [19 - 29]. A crossbar interconnection system is typically used in an environment where processors share L2 cache banks. It allows multiple core ports to launch operations to the L2 subsystem and receiving data or getting invalidates from L2 in the same cycle [19]. In general, a crossbar has three busses: Address Bus, Data in bus and Data out bus. Data out bus is used for writebacks from each core to the banks and data in bus is used for data reload and to invalidate addresses from all L2 banks to the cores.

2.2.3 Shared L2 and L3 Cache

In a shared L2 cache architecture, each processor has its own private L1 cache sharing an L2 cache as shown in Figure 8a. In a shared L3 bus architecture, each processor has its own private L1 and L2 cache sharing an L3 cache as shown in Figure 8b.

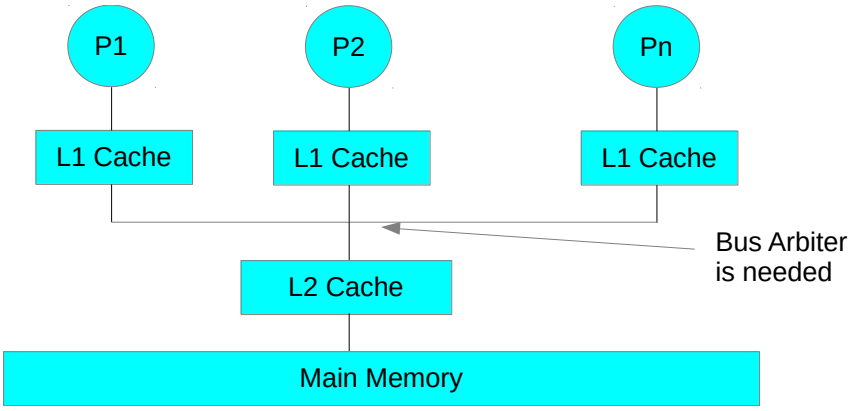

Figure 8a. Shared L2 bus

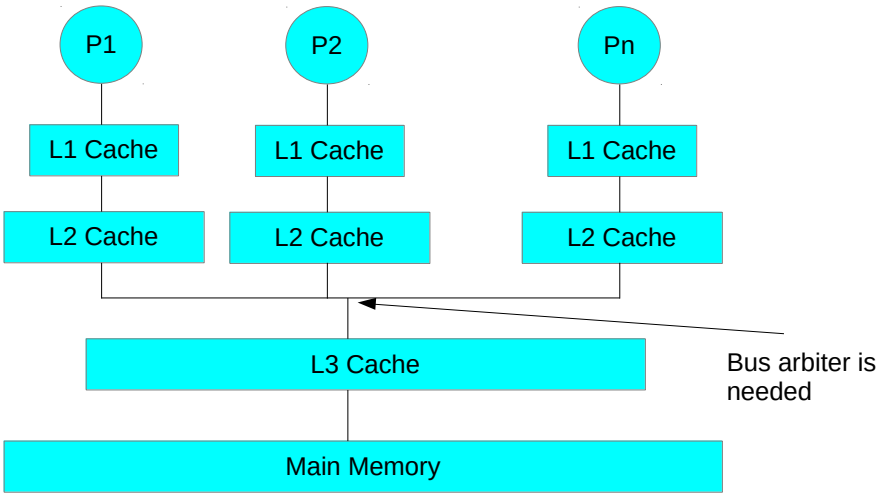

Figure 8b. Shared L3 bus

These architectures are widely used in industries due to the ease of implementation. A bus arbiter is needed to satisfy all requests between L1 caches and the L2 cache for the shared L2 configuration and between L2, and L3 caches for the shared L3 configuration. As the number of processor increases, the bus leading to the shared L2 and L3 caches will get more congested, hence reducing the performance of the overall system. These architectures are evaluated in Chapter 3 and 4.

2.2.4 Ring Topology

Ring topology consists of placing caches in a ring shaped manner. The idea behind a ring topology is to reduce message passing on a shared bus while maintaining cache coherency by passing messages from one core to another in a systematic way. Recent processor researchers use these ideas to come up with better and faster multiprocessors [30-32]. Figure 9 shows the Ring topology in general.

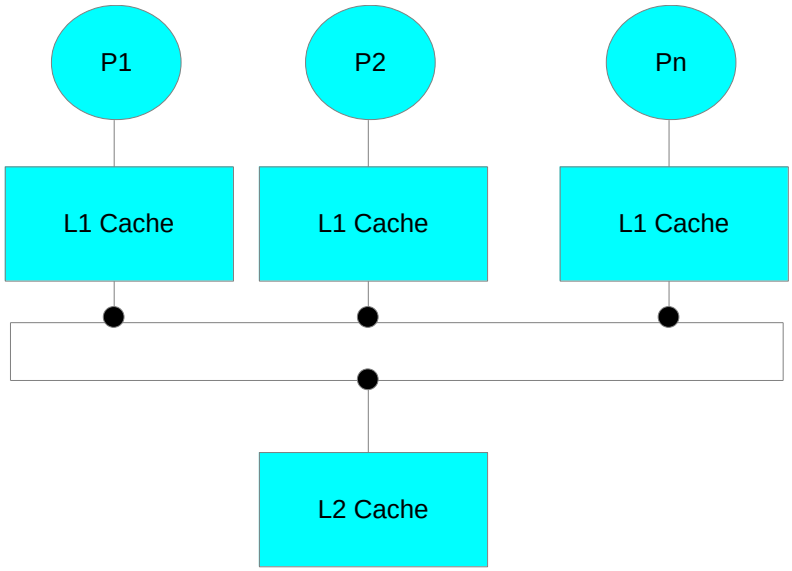

Figure 9. Ring Topology

2.3 Cache Architecture Performance Evaluation Tools

From the early 1990s until today, researchers have been trying to find the best cache architecture that delivers the best overall performance (higher hit rate and lower miss rate) for multiprocessor [33 - 44]. As the number of processor increases, the cache architecture can become very complex. Cache architectures such as single shared L2 bus, hierarchical bus, and ring-shaped architecture are widely known and studied independently [33 - 45]. The lack of any publicly available tools to evaluate the performance of these various cache architectures has prompted the development of a

new simulation tool, Abakus, which can be used to emulate and study different cache hierarchies and configurations. There a three simulation tools that are used in academia as of January 2012 and they are: SimpleScalar, Simics and SystemC. We choose not to use SimpleScalar because it lacks the ability to handle cache arbitration, and multithreading for future development. Simics does not provide users the ability to edit some of the configuration files. SystemC is recognized by IEEE as a standard for system-level modeling, design and verification. SystemC has a close resemblance to Verilog/VHDL, a hardware language; hence it can be used to model any hardware unlike SimpleScalar that focuses on one specific class of hardware architecture and a major reason why we choose SystemC.

Examples on how to emulate a cache hierarchy and configuration using Abakus may be found in Appendix B. Dr. Louis Johnson is the creator and in charge of updating and maintaining Abakus. Julius Marpaung is in charge of cross compiling benchmarks to MIPS that are used by Abakus, running the benchmarks and checking the results to make sure that they are consistent with the output reference provided by SPEC CPU2006. As of April 2012, Abakus does not have the ability to run multithreaded benchmarks hence no coherency or consistency model needs to be developed for processors to interact with each other.

CHAPTER III

ABAKUS AND VALIDATION

3.1 Using Abakus

Abakus uses SystemC 2.2.0 from http://www.accellera.org that is widely used to model hardware. The current processor model used in Abakus is a MIPS scalar processor; hence it runs on the MIPS instruction set. To cross compile any C/C++ based benchmarks to MIPS using Crosstool from http://www.kegel.com/crosstool, please refer to Appendix A. Alternatively, you can also use uClibc from http://uclibc.org but the procedure will not be covered in this dissertation. As of January 2012, Crosstool and uClibc are widely used in academia, but they do not have the ability to fully support multithreading with OpenMP yet, so the older pthreads library must be used. The system calls for pthreads have not been added to abacus so that only single threaded benchmarks can be run with abacus. The standard linux system calls from SimIt-MIPS have been added to Abakus in order to work with SPEC CPU2006 benchmarks. We also add to SimIt-MIPS the capability to generate an instruction trace file.

In order to verify whether or not Abakus is correctly executing the program/benchmark, Abakus compares the instructions it executes and the register contents with the trace file. Instructions on how to use and download SimIt-MIPS may be found at http://simit-mips.sourceforge.net.

3.2 Validation

SPEC CPU2006 benchmark papers [46 – 53] are used as a guideline to validate the performance of our simulation, Abakus. These papers use the Intel and AMD instruction set while Abakus uses MIPS instruction set. The difference between Intel, AMD and MIPS instruction set is beyond the scope of this dissertation and will not be discussed. Figure 10 shows the discrepancies in the total number of instructions to run the full simulation using various processor configurations and simulation tools shown in Figure 11. Some discrepancies/differences are expected when comparing the performance of the non MIPS architecture to MIPS architecture; however, even discrepancies/differences are found when comparing the results among Intel processors as shown in Figure 12 and 13 where MPKI stands for Misses Per Kilo Instructions. All simulations done using Abakus in this dissertation are limited to 1 billion instructions due to the amount of time needed to run those simulations, hence that is another reason why there are some discrepancies between the results from Abakus and others [49-56]. There are five benchmarks that can be cross compiled into MIPS from SPEC CPU2006 and they are sjeng, bzip, mcf, libq, and specrand.

Karthik Ganesan		Arun Nair		Tribuvan Prakash	
Benchmark	# of instruction (Billion)	Benchmark	# of instruction (Billion)	Benchmark	# of instruction (Billion)
		Bzip2-combined	371.92	Bzip2-combined	332
Bzip2	213.9			Bzip2-program	553
sjeng	3187.7	Sjeng	2654.13	sjeng	2351
libquantum	1989	libquantum	4534.27	libquantum	4013
MCF	Not Available	MCF	464.98	mcf	322

Figure 10. Total Number of Instructions for some benchmarks

Author	Julius Marpaung	Shengmei Li		Arun Nair
Simulation Tool	SystemC	Not Used	Not Used	Pinpoint + SimPoint
Core	Intel® Xeon® 2.66 GHz (Quad Core)	Intel® Xeon® 5160 3GHz (Dual Core)	AMD® Opteron® 2222 SE 3 GHz (Dual Core)	Intel® Pentium 4 2.8 GHz
L1 Cache	256 KB	32 KB (data) + 32 KB (instruction), 8 way	64 KB (data) + 64 KB (instruction), 2 way	16 KB (data) + 16 KB (instruction), 8 way
L2 Cache	2 x 6MB	4 MB (shared by 2 cores), 16 way	1 MB (private), 16 way	1 MB, 8 way
Cross Compiler	Crosstool -> MIPS instruction	Not used	Not used	Not used
Length of Run	1 Billion instruction	Full Run	Full Run	Full Run
Author	Sarah Bird	Tribuvan Prakash	Aamer Jaleel	
Simulation Tool	Not Used	Not Used	CMP$im	
Core	Intel® Xeon® 5160 3 GHz (Dual Core)	Intel® Core 2 Duo® E6400 2.13GHz (Dual Core)	Intel® Xeon® 2.8 GHz	
L1 Cache	32 KB (data) + 32 KB (instruction)	32 KB (data) + 32 KB (instruction), 8 way	32 KB (data) + 32 KB (instruction)	
L2 Cache	4 MB (shared)	2MB (shared), 8-way	512 KB	
Cross Compiler	Not used	Not used	Not used	
Length of Run	Full Run	Full Run	Full Run	

Figure 11. Various setups for simulation

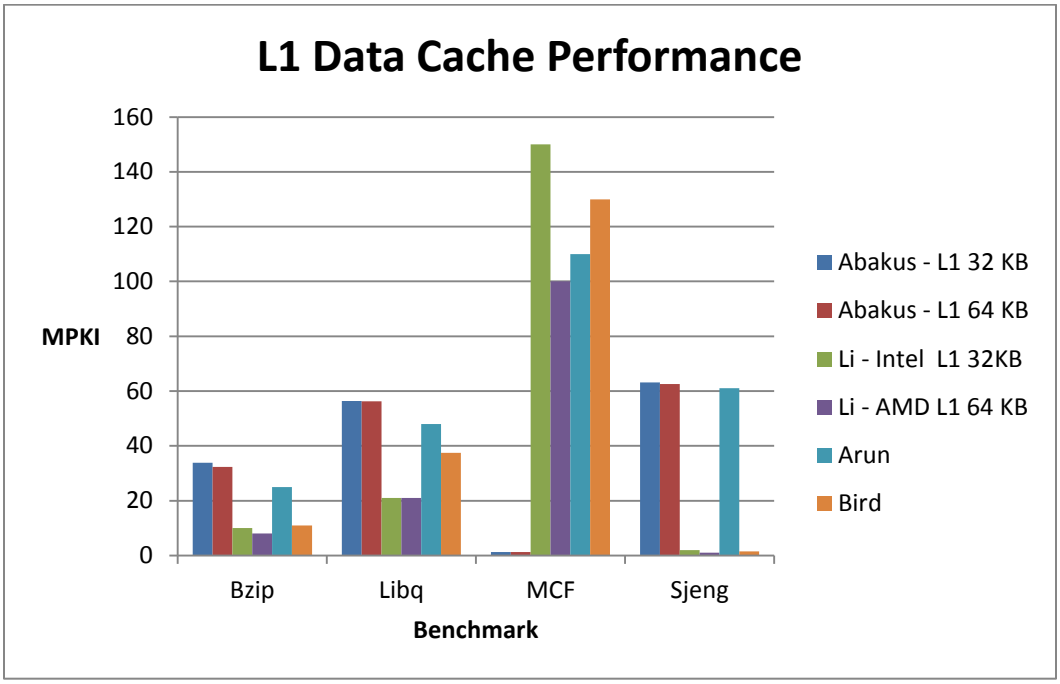

Figure 12.Comparison of L1 Data Cache Performance between Abakus, Li, Arun, and Bird

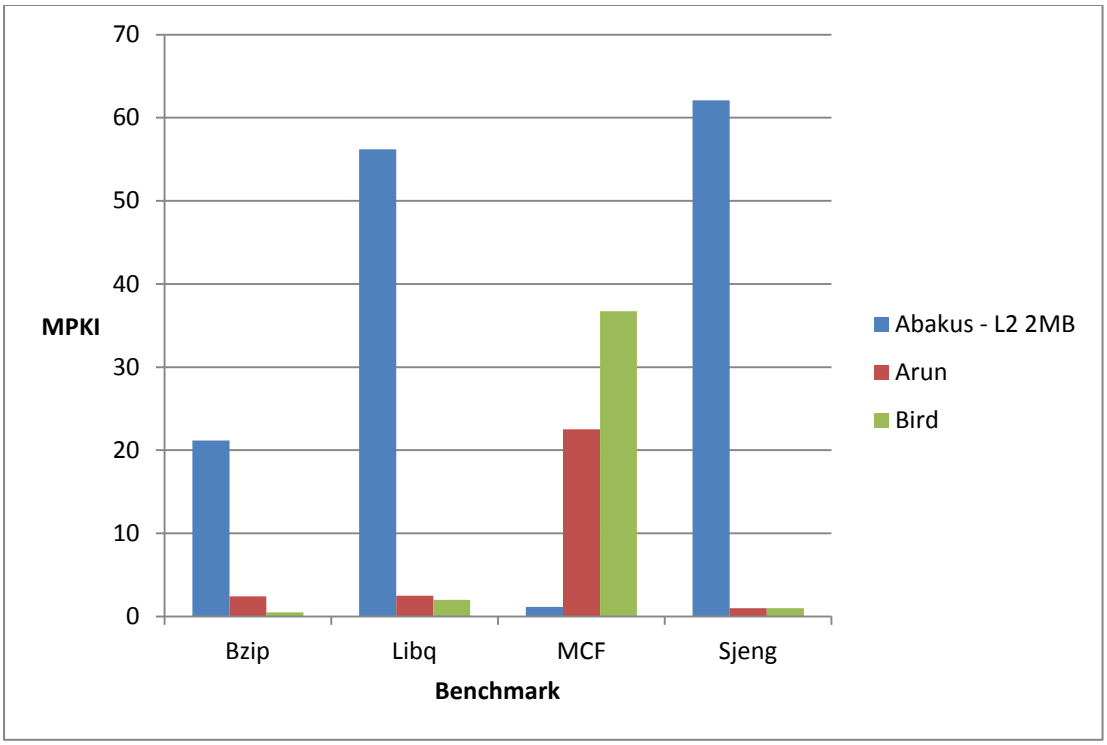

Figure 13.Comparison of L2 Cache Performance between Abakus, Arun, and Bird

Jaleel[49] configures his simulation to support the following data and instruction cache configurations: 1-way 32KB, 2-way 64KB, 4-way 128KB, 8-way 256KB up to 2048-way 128MB; using 8-way 256KB L2 cache. Jaleel shows that any instruction cache from 32KB and beyond will result in virtually zero miss rates for the instruction. To compare the results shown by Jaleel with Abakus, we need to use the following formula:

(# of misses / # of simulated instructions) x total # of instruction = # of misses / program [1]

Figure 14 – 18 show the reconstructed number of misses per program from Jaleel and Abakus for sjeng, bzip, mcf, libq, and specrand using 32 KB instruction cache, 32KB to 128 MB data cache and 8-way 256KB L2 cache.

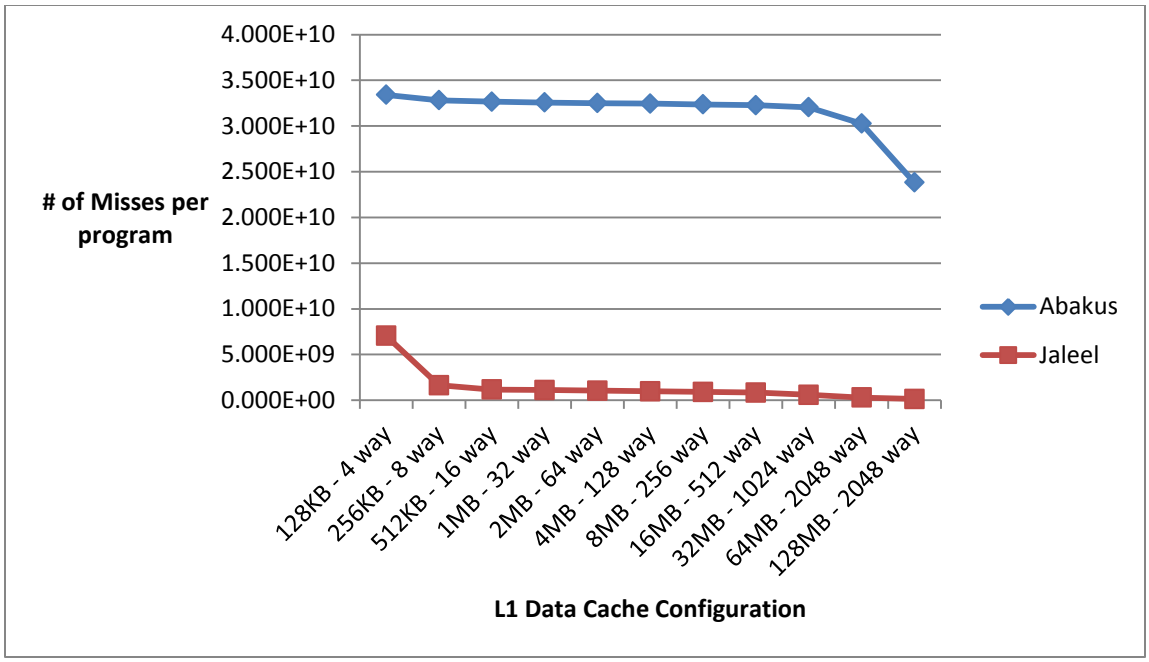

Figure 14. Number of misses per program constructed from Jaleel and Abakus for Sjeng

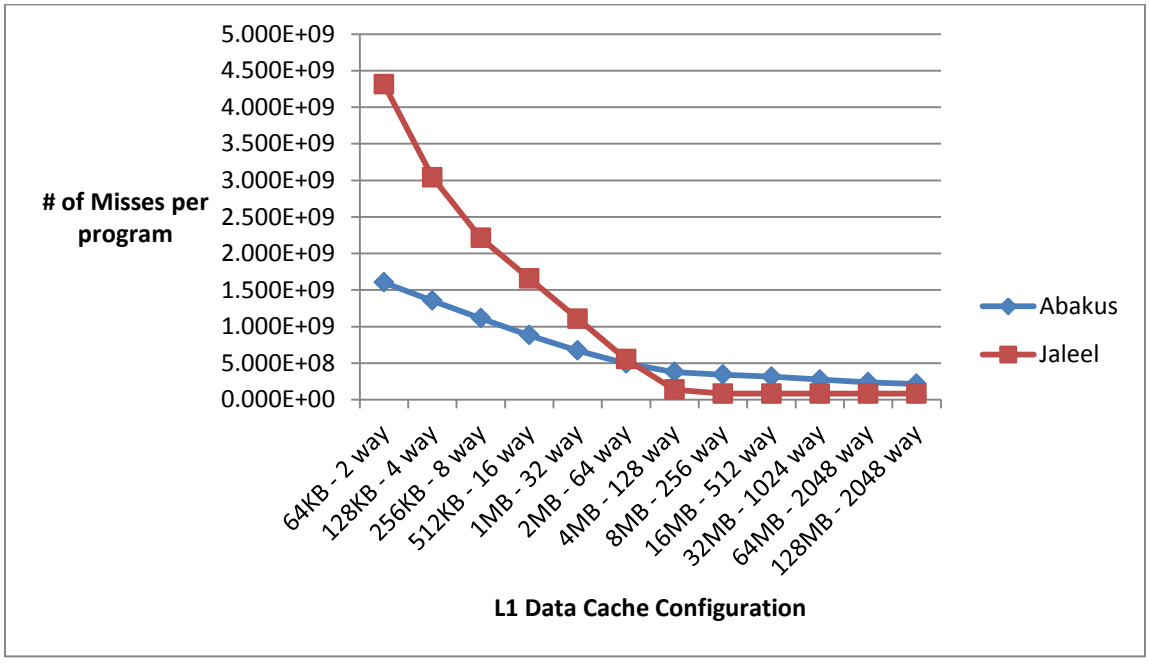

Figure 15. Number of misses per program constructed from Jaleel and Abakus for Bzip

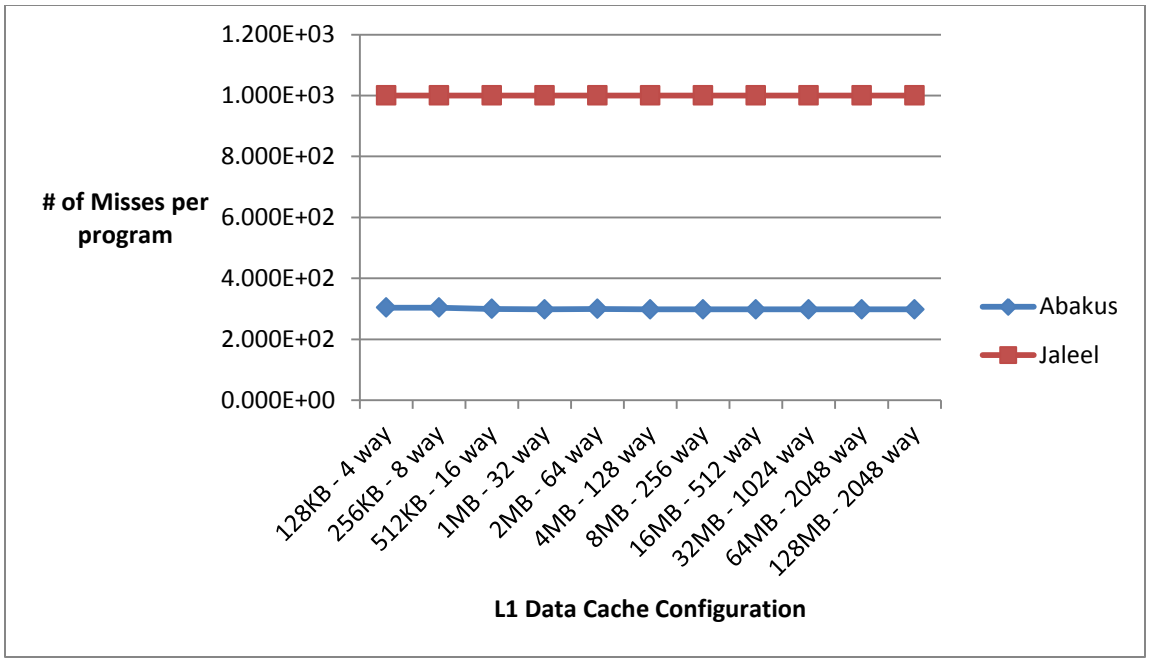

Figure 16. Number of misses per program constructed from Jaleel and Abakus for Specrand

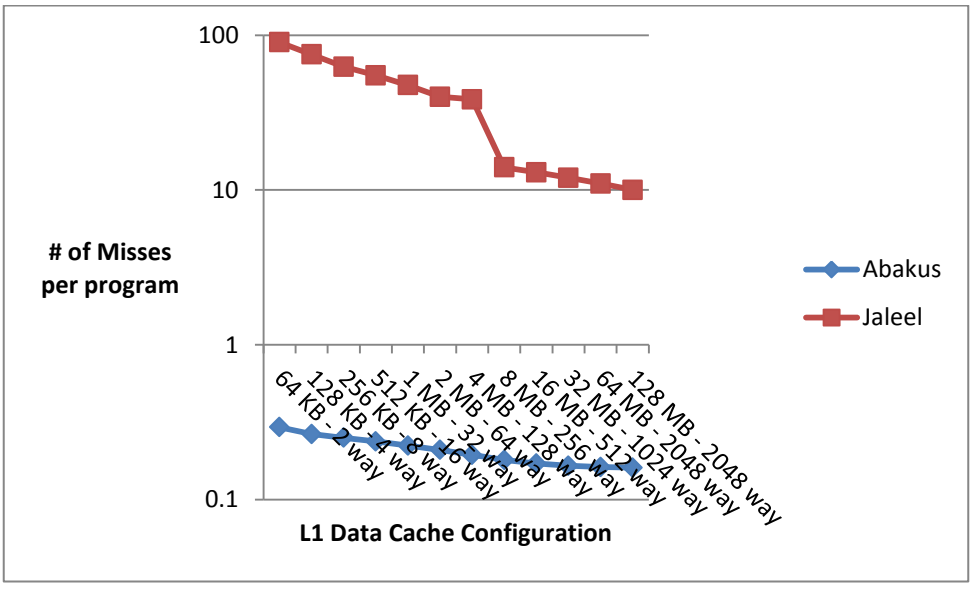

Figure 17. Number of misses per program constructed from Jaleel and Abakus for MCF on a logarithmic scale

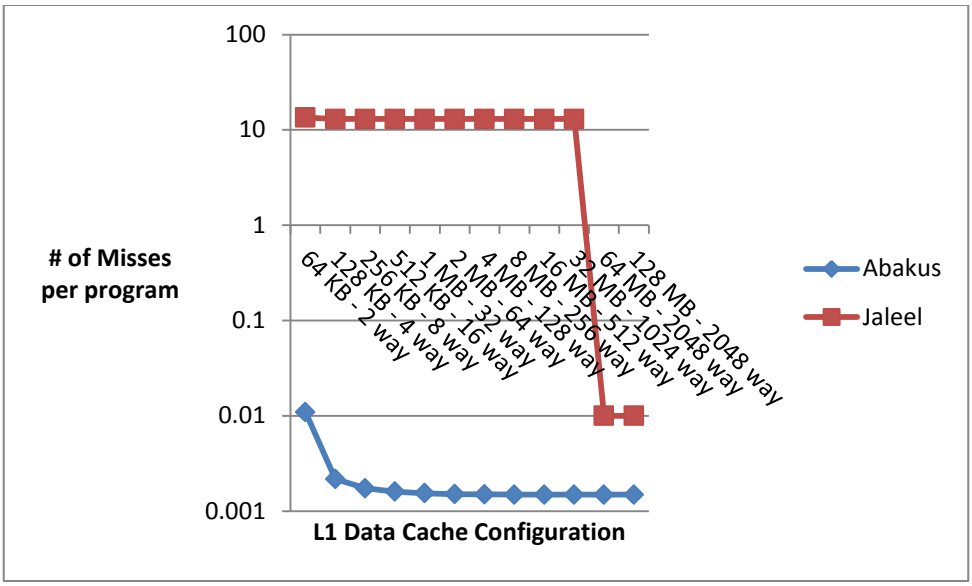

Figure 18. Number of misses per program constructed from Jaleel and Abakus for Libquantum on a logarithmic scale

As expected and shown in Figure 14 – 18, we have some discrepancies between the results from Abakus and Jaleel due to the difference in the instruction set and the number of instructions executed.

Lu Peng et all [48] published a paper regarding the execution time for SPEC CPU2006 benchmarks. They used a dual-core processor to run the benchmarks without using any simulation tools. Their Intel Core 2 results are compared with our dual-core shared L2 results, and their Pentium D 830 and AMD Athlon 64 results are compared with our dual-core shared L3 results due to the cache configuration for the processors. In order to find Lu Peng's number of clock cycles needed to run their benchmarks into full completion, we need to multiply the amount of execution time in seconds by the processor clock frequency. In order to estimate the number of clock cycles needed to run our benchmarks into full completion we need to divide the total number of instructions for each benchmark by 1 billion instructions and multiply the result by the

number of recorded clock cycles for 1 billion instructions. Figure 19 shows the comparison of the number of clock cycles to run benchmarks to full completion between Abakus' and Lu Peng's.

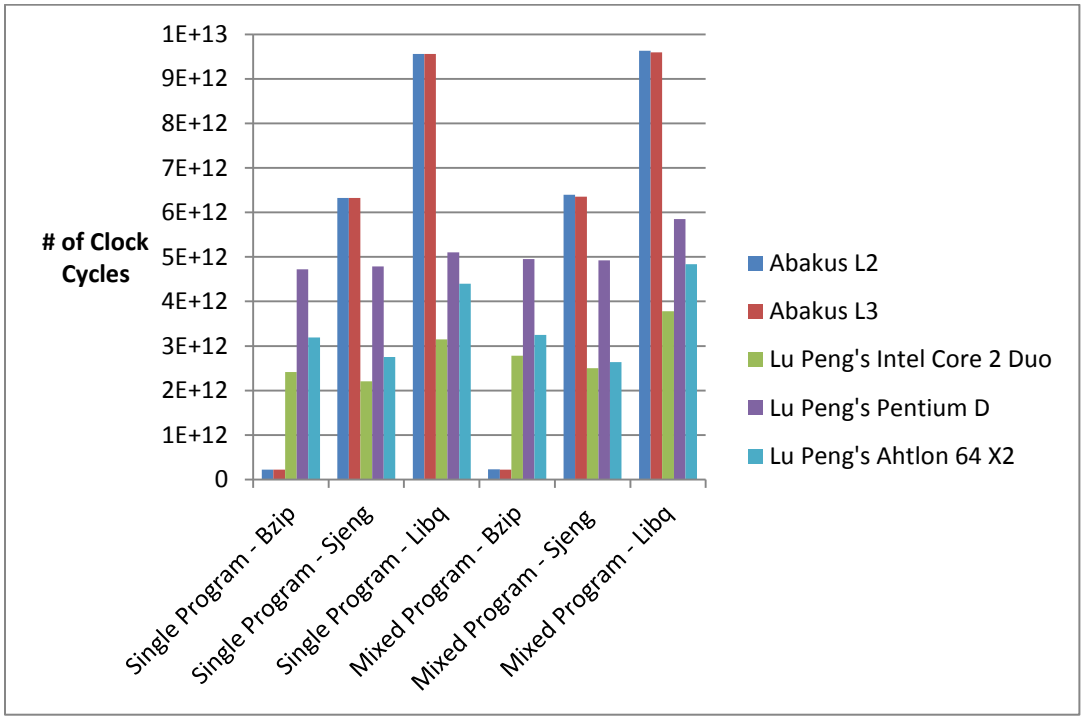

Figure 19. Number of clock cycles for Shared L2 and L3 architectures

As shown in Figure 19, our results are similar to Lu Peng's with differences due to the difference in instruction set and processor model used to run the benchmarks.

CHAPTER IV

PERFORMANCE LIMITATION OF SHARED BUS MULTICORE

In this chapter, we discuss the performance of the following multi-core architectures: Dual-Core Shared L2, Quad-Core Shared L2, Octal-Core Shared L2, 16-Core Shared L2, Dual-Core Shared L3, Quad-Core Shared L3, Octal-Core Shared L3, 16-Core Shared L3, Quad-Core Hierarchy, Octal-Core Hierarchy, and 16-Core Hierarchy. Four SPEC CPU 2006 benchmarks are used: Bzip, MCF, Libq, and Sjeng.

4.1 Dual-Core Shared L2

The Dual-Core Shared L2 architecture is shown in Figure 20. The recorded Average IPC, L2 Miss Rate, Bus 1 Busy Rate, Bus 1 Wait Rate, Bus 2 Busy Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 21 – 25 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

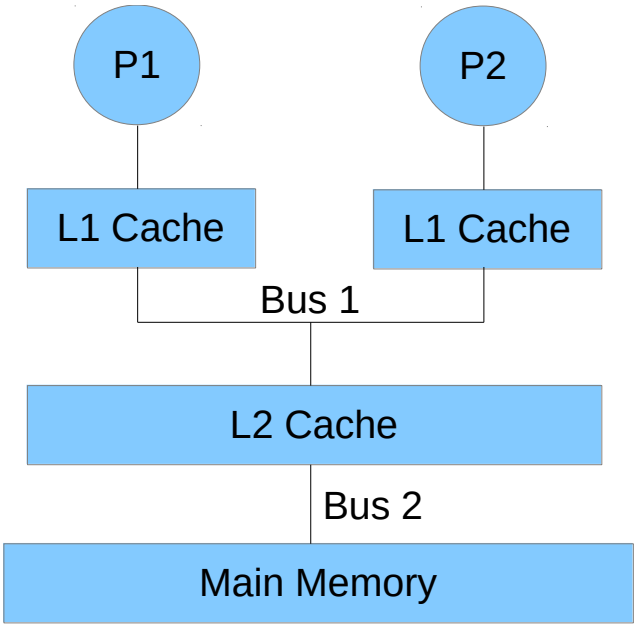

Figure 20. Dual-Core Shared L2 Architecture

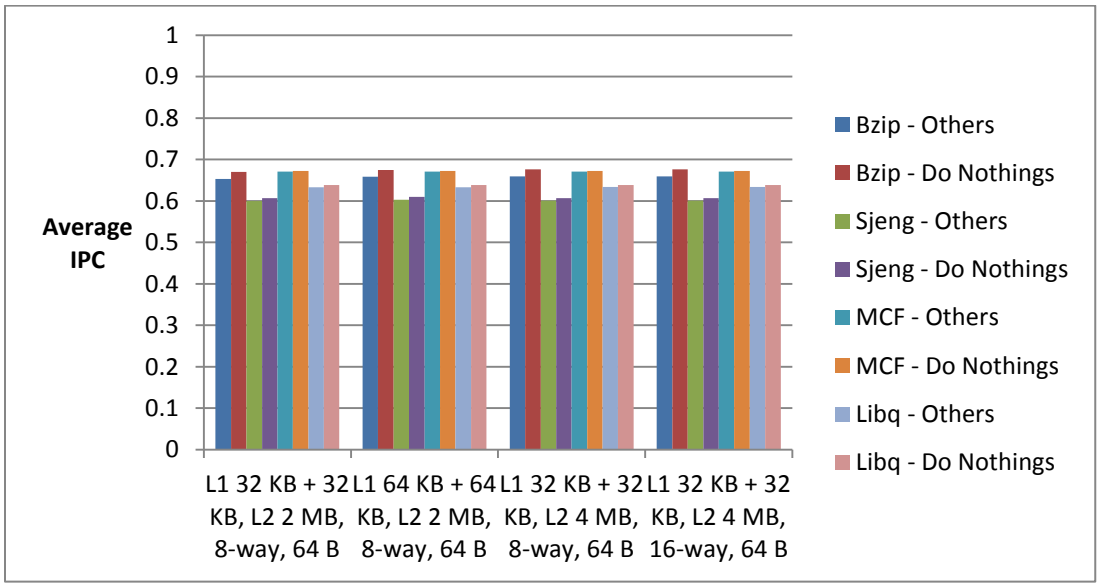

Figure 21. Average IPC for Dual-Core Shared L2

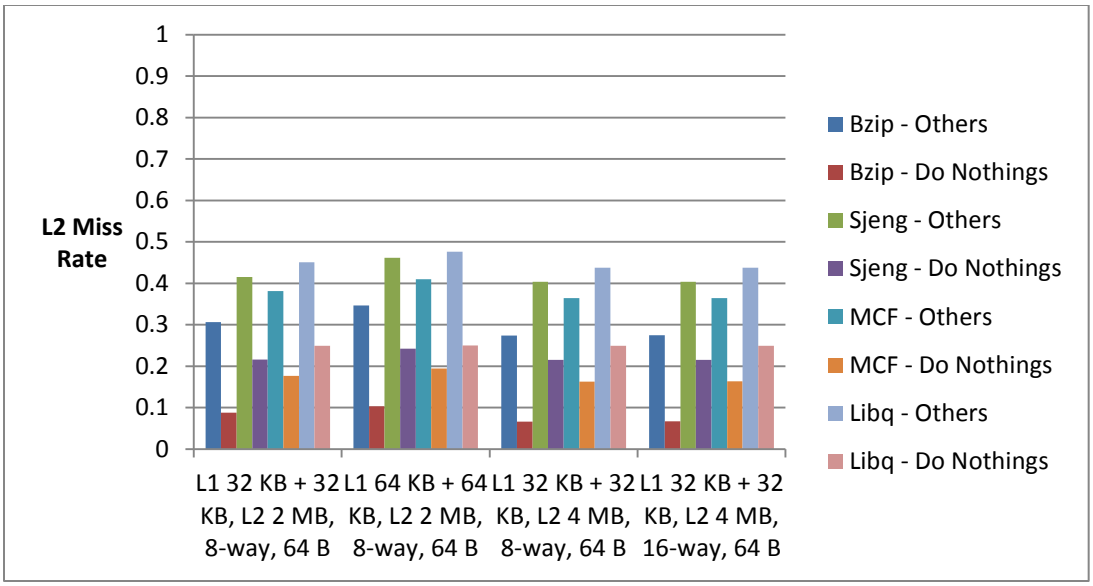

Figure 22. L2 Miss Rate for Dual-Core Shared L2

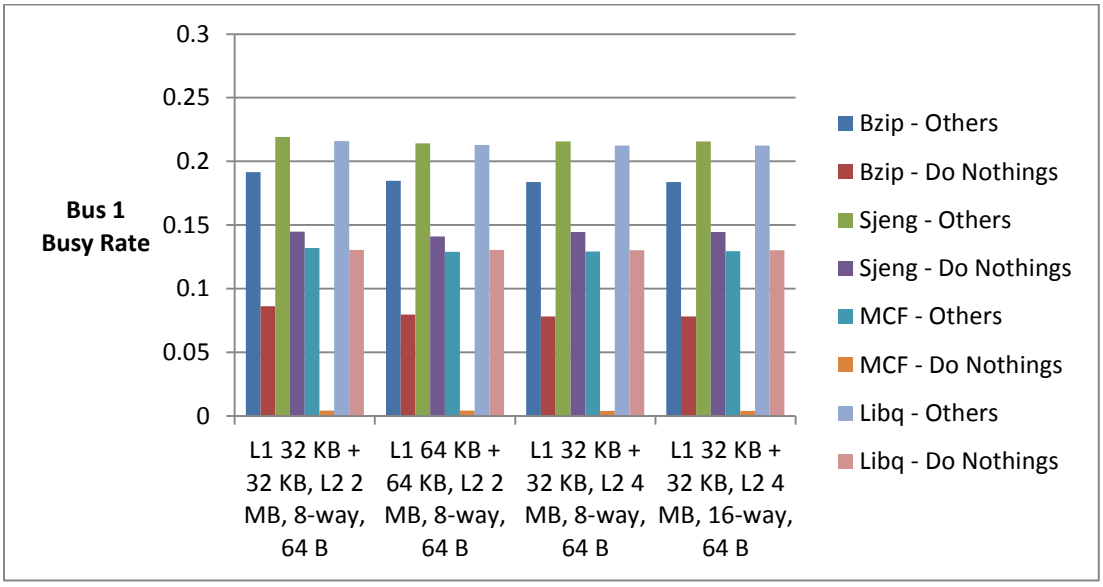

Figure 23. Bus 1 Busy Rate for Dual-Core Shared L2

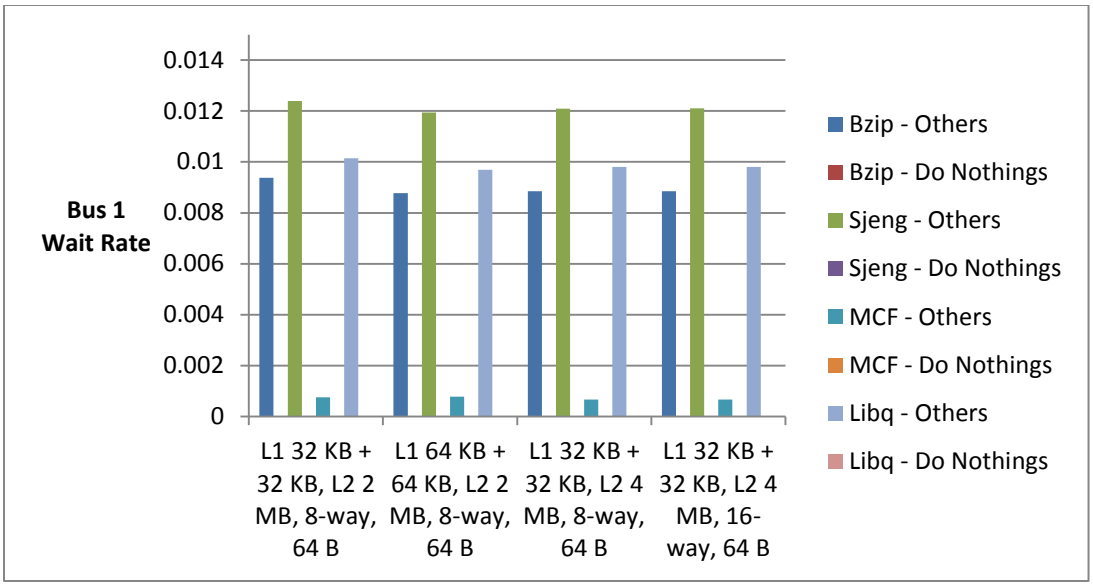

Figure 24. Bus 1 Wait Rate for Dual-Core Shared L2

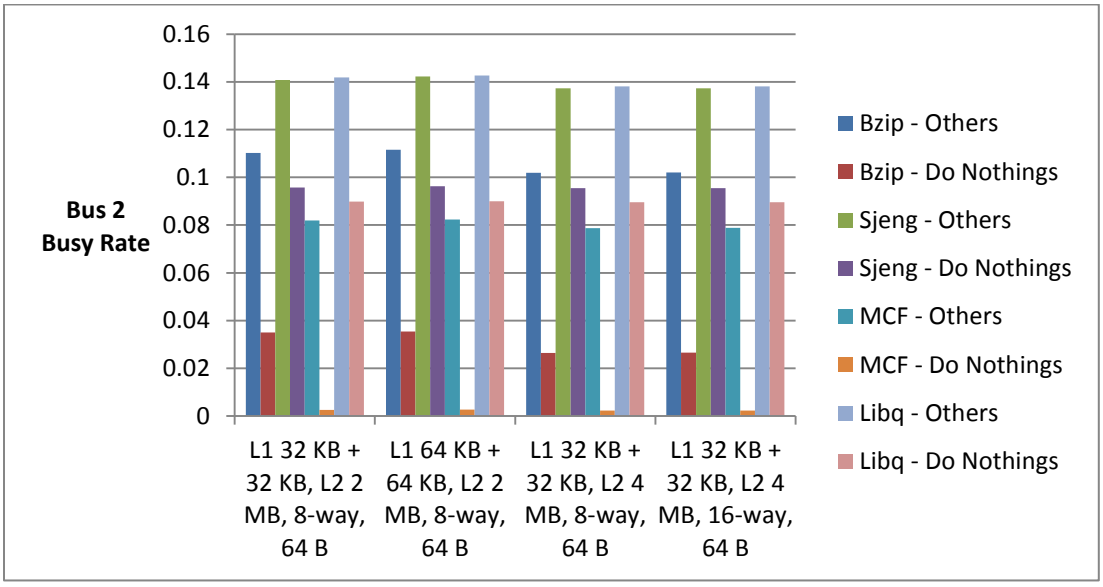

Figure 25. Bus 2 Busy Rate for Dual-Core Shared L2

The L2 miss rate is defined as the number of misses per access in L2. The bus 1 and bus 2 busy rate is defined as the probability that a request has been granted in a given clock cycle on bus 1 and bus 2 respectively. Bus 1 wait rate is defined as the probability that a processor has ungranted requests in a given clock cycle on bus 1.

As shown in Figure 21 - 25, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 1 busy rate, bus 1 wait rate and bus 2 busy rate. As expected, the bus 2 contention is fairly low because bus 2 only needs to handle 2 processors. Chapter 4.2 will discuss the impact of using four processors using a shared L2 cache.

4.2 Quad-Core Shared L2

The Quad-Core Shared L2 architecture is shown in Figure 26. The recorded Average IPC, L2 Miss Rate, Bus 1 Busy Rate, Bus 1 Wait Rate, Bus 2 Busy Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 27 – 31 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

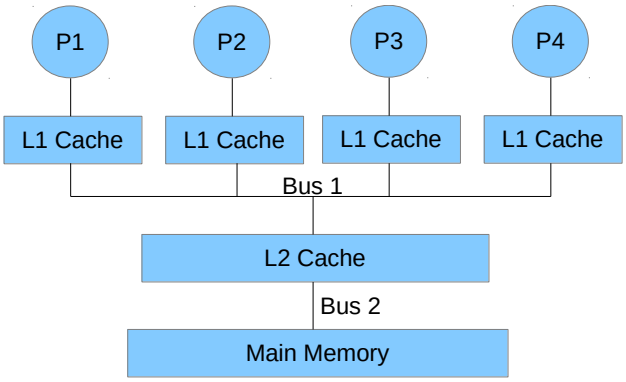

Figure 26. Quad-Core Shared L2 Architecture

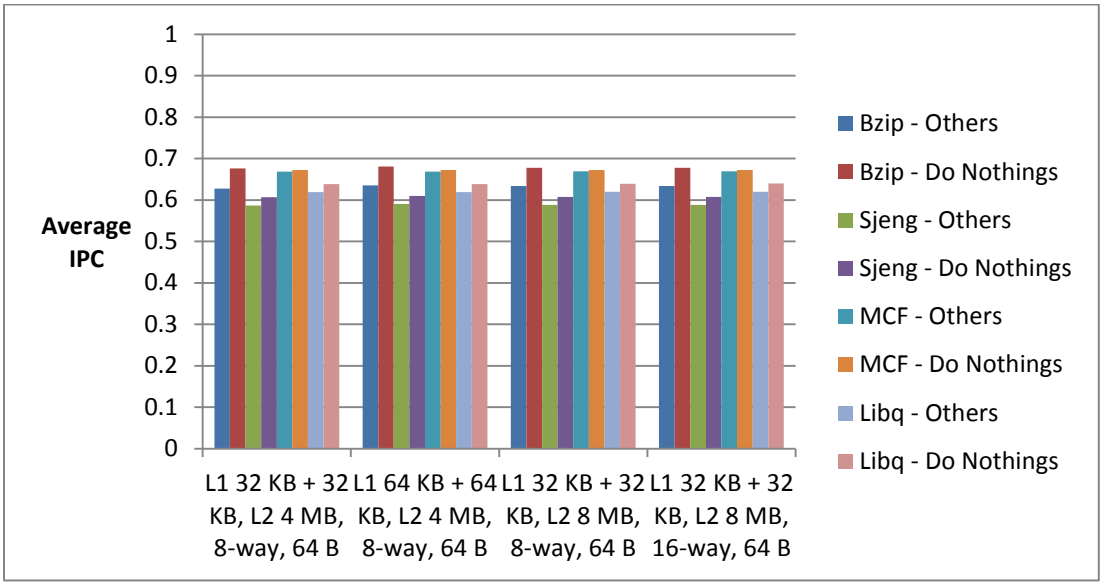

Figure 27. Average IPC for Quad-Core Shared L2

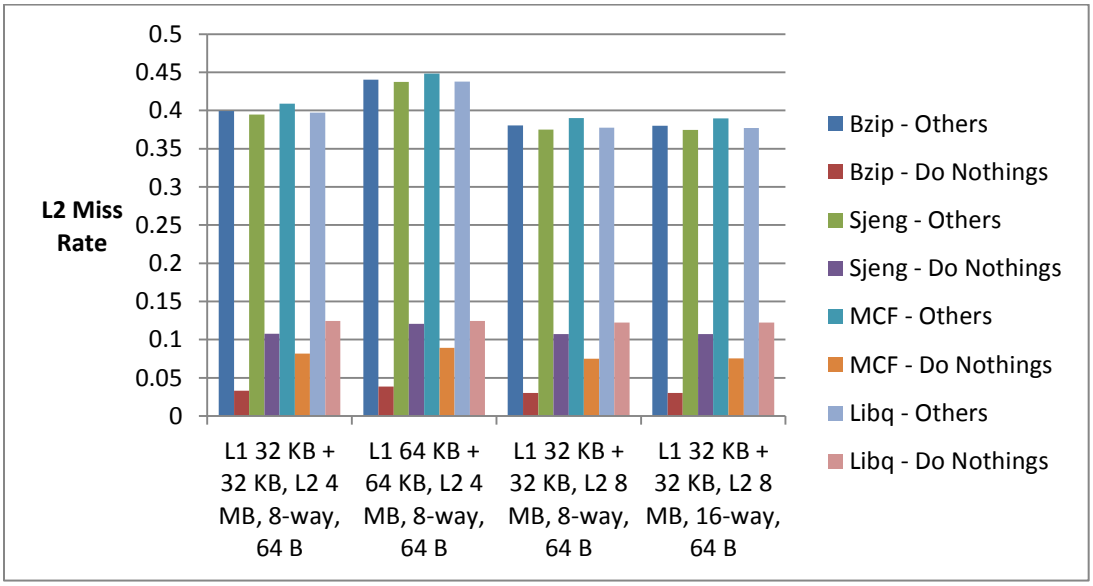

Figure 28. L2 Miss Rate for Quad-Core Shared L2

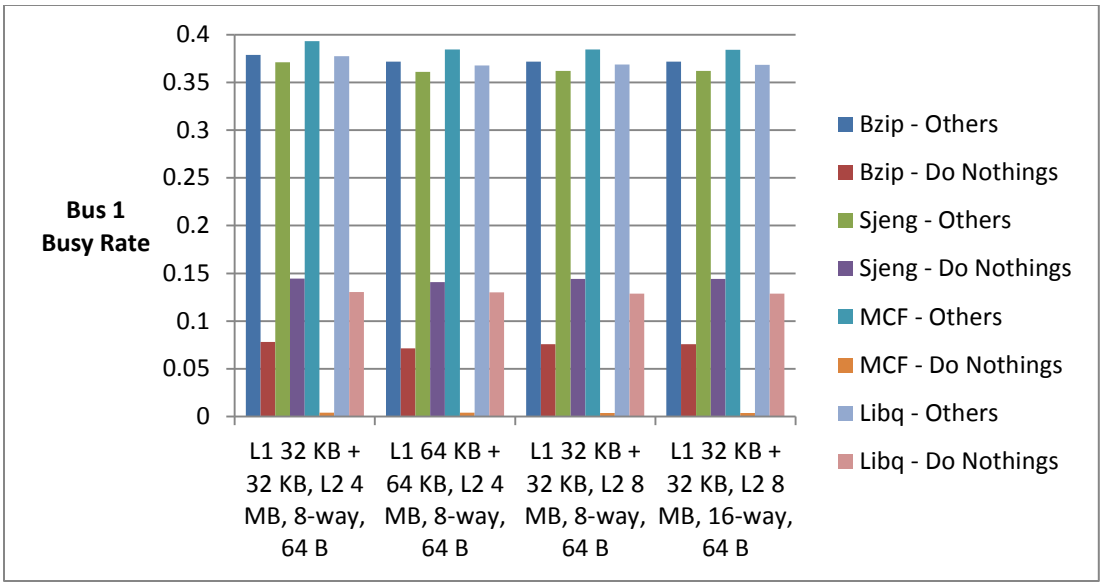

Figure 29. Bus 1 Busy Rate for Quad-Core Shared L2

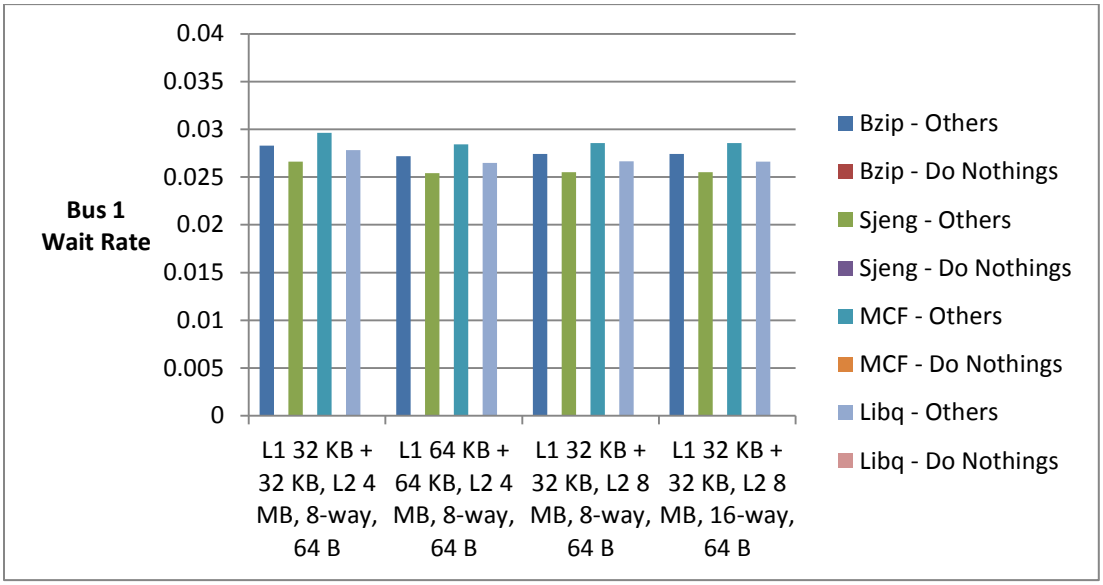

Figure 30. Bus 1 Wait Rate for Quad-Core Shared L2

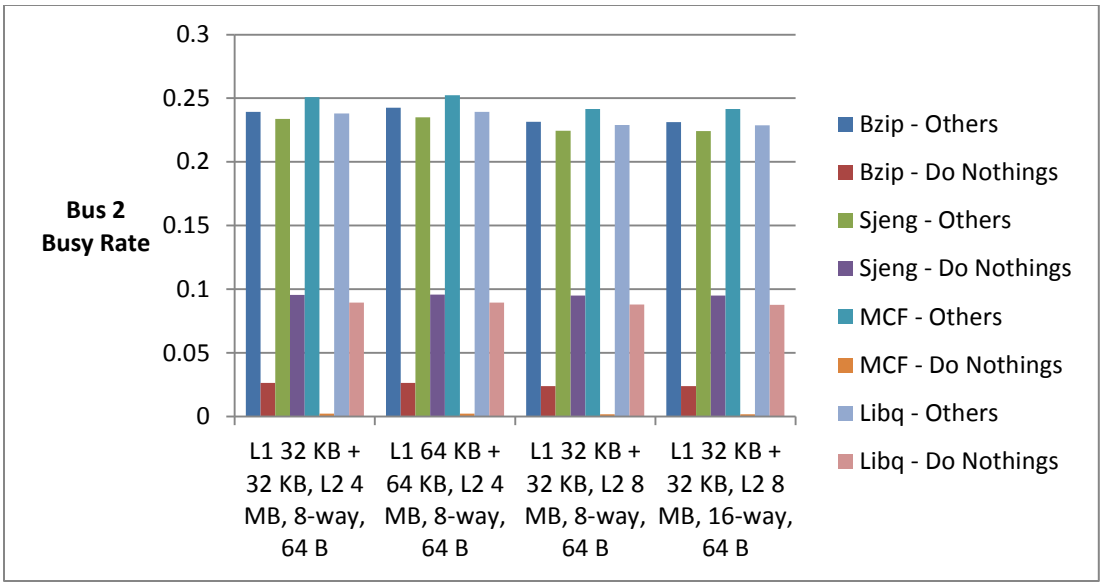

Figure 31. Bus 2 Busy Rate for Quad-Core Shared L2

As shown in Figure 27 - 31, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 1 busy rate, bus 1 wait rate and bus 2 busy rate. The bus 1 and 2 contentions in general are almost doubled compared to the Dual-Core shared L2 result. Compared to the Dual-Core Shared L2 result, the L2 miss rate does not change that much. The issue that needs to be addressed later in the chapter is how many processors a shared L2 bus can handle before the bus 1 and 2 contentions go to 100%. Chapter 4.3 will discuss the impact of using eight processors using a shared L2 cache.

4.3 Octal-Core Shared L2

The Octal-Core Shared L2 architecture is shown in Figure 32. The recorded Average IPC, L2 Miss Rate, Bus 1 Busy Rate, Bus 1 Wait Rate, Bus 2 Busy Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 33 –

37 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

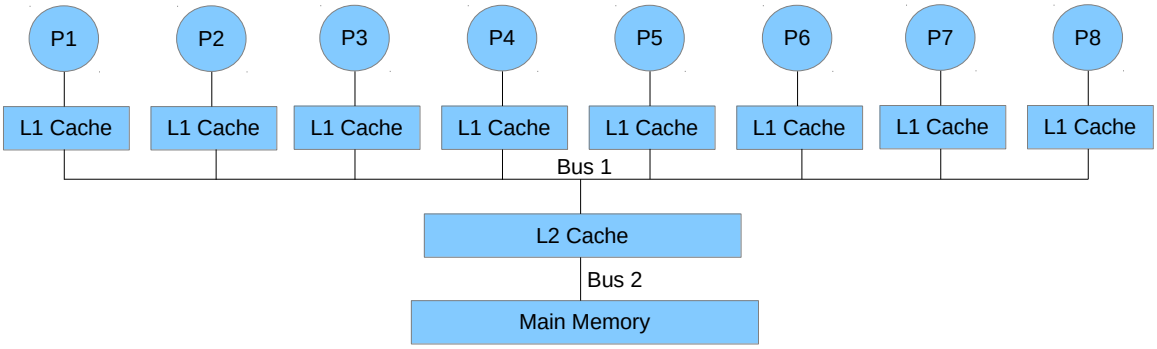

Figure 32. Octal-Core Shared L2 Architecture

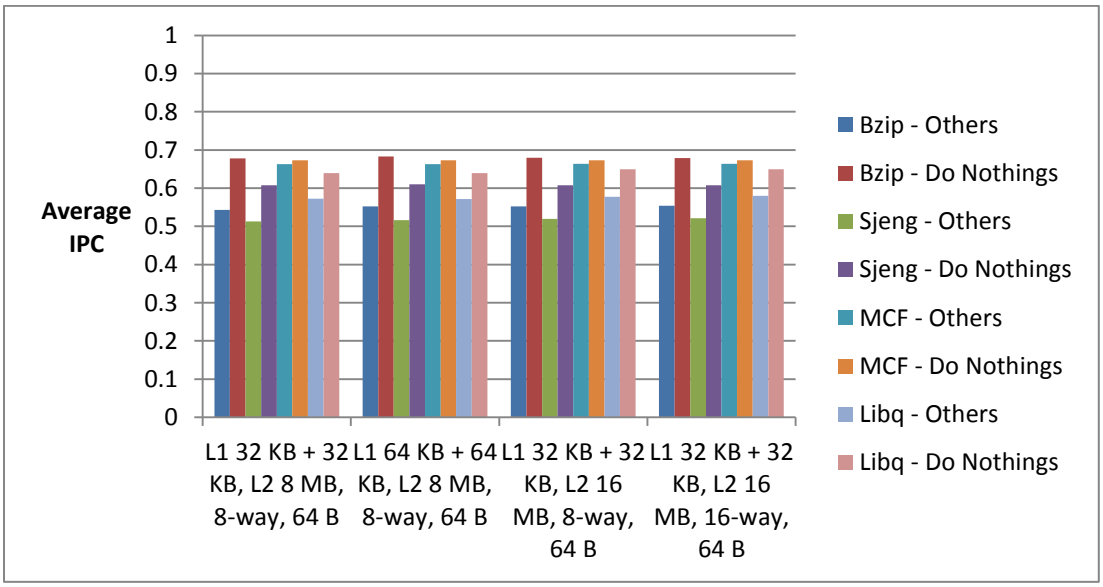

Figure 33. Average IPC for Octal-Core Shared L2

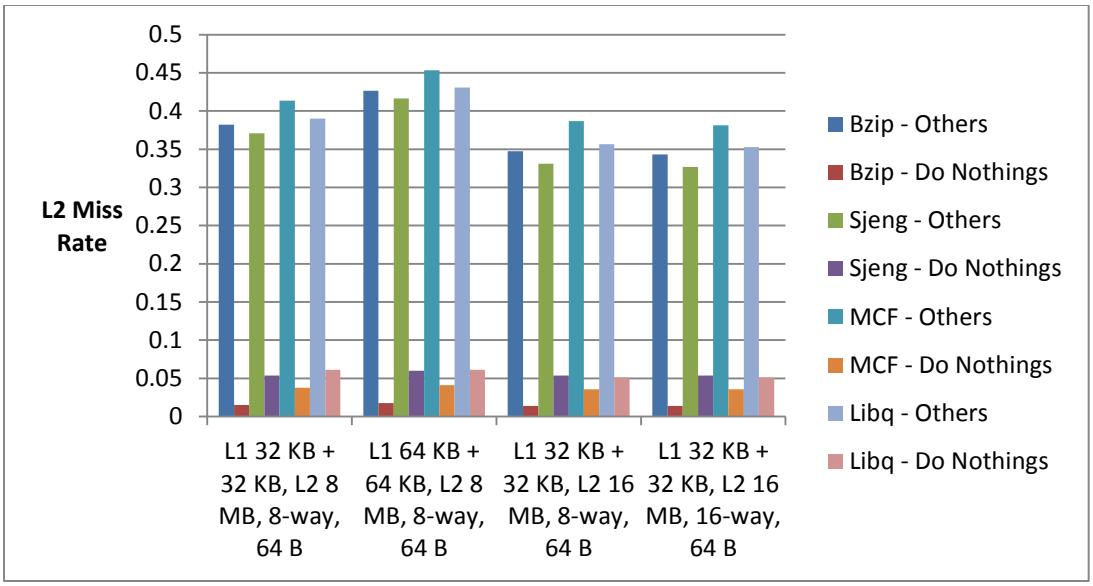

Figure 34. L2 Miss Rate for Octal-Core Shared L2

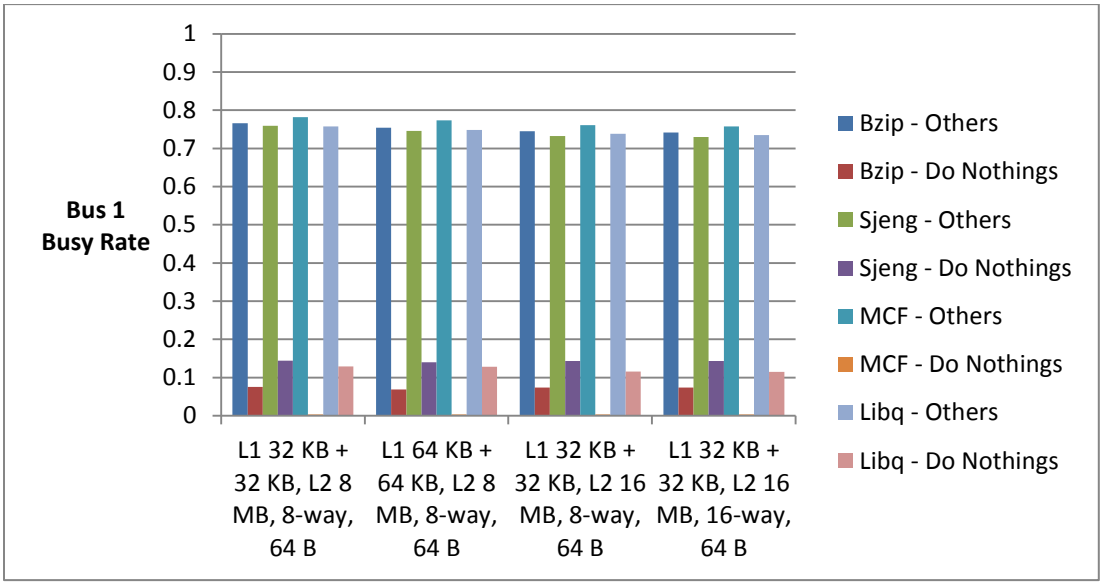

Figure 35. Bus 1 Busy Rate for Octal-Core Shared L2

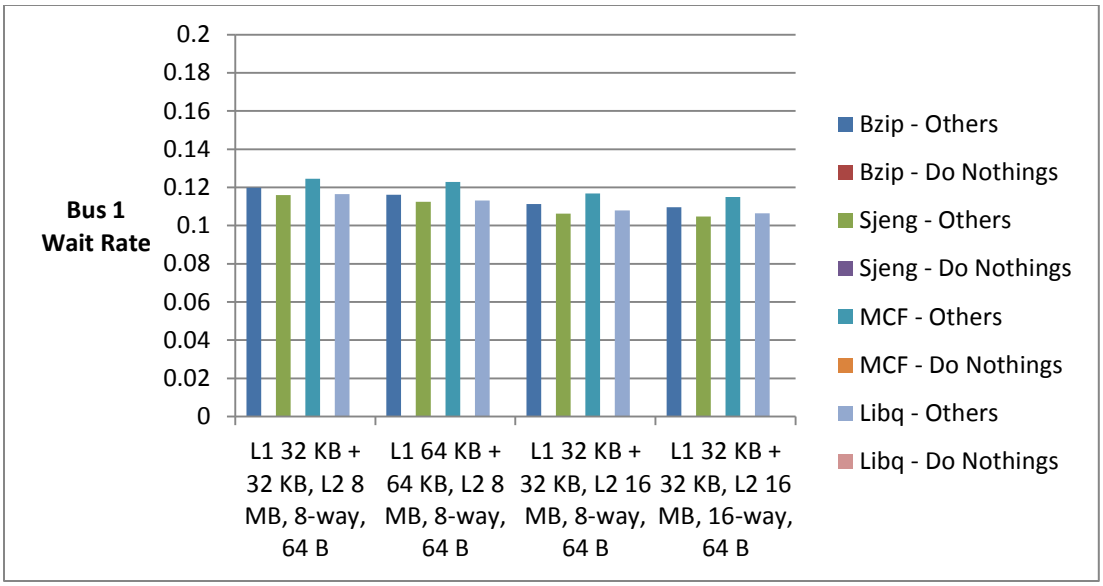

Figure 36. Bus 1 Wait Rate for Octal-Core Shared L2

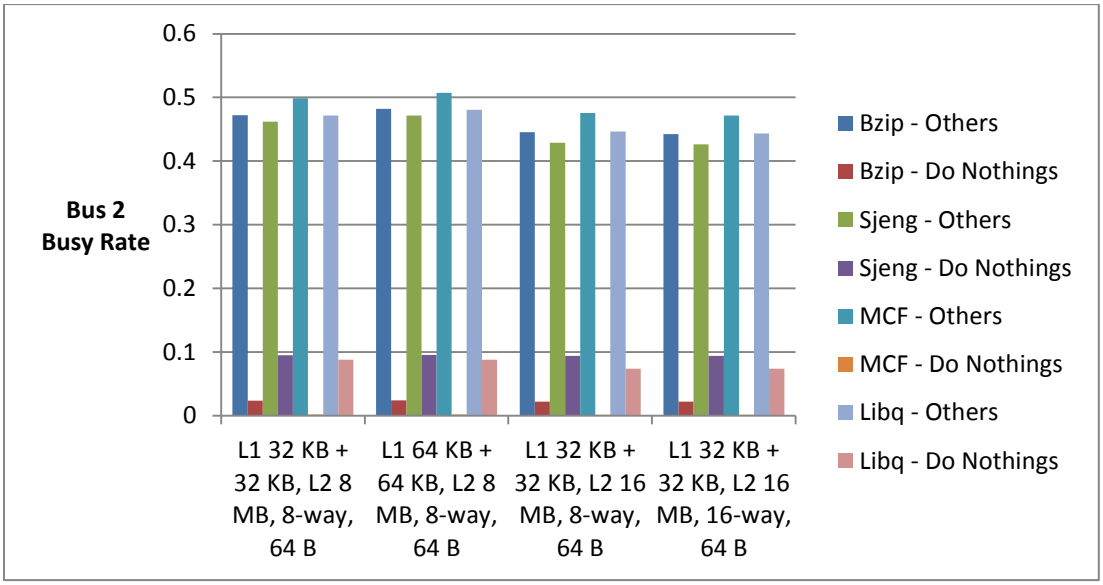

Figure 37. Bus 2 Busy Rate for Octal-Core Shared L2

As shown in Figure 33 - 37, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 1 busy rate, bus 1 wait rate and bus 2 busy rate. The bus 1 and 2 contentions in general are almost doubled compared to the Quad-Core shared L2 result. Compared to the Quad-Core Shared L2 result, the L2 miss rate does not change. Chapter 4.4 will discuss the impact of using sixteen processors using a shared L2 cache.

4.4 16-Core Shared L2

The 16-Core Shared L2 architecture is shown in Figure 38. The recorded Average IPC, L2 Miss Rate, Bus 1 Busy Rate, Bus 1 Wait Rate, Bus 2 Busy Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 39 – 43 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

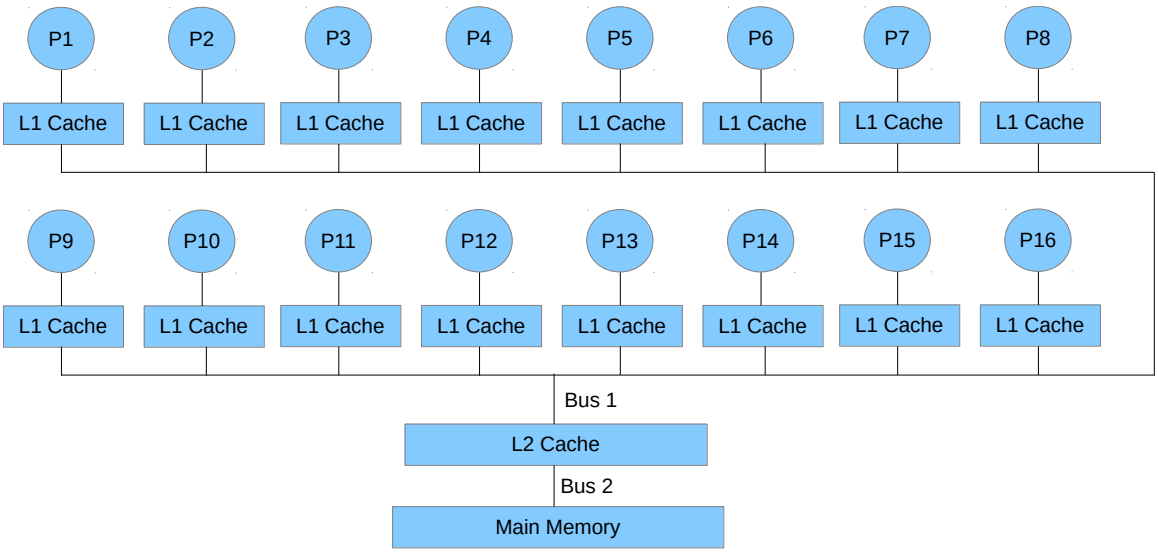

Figure 38. 16-Core Shared L2 Architecture

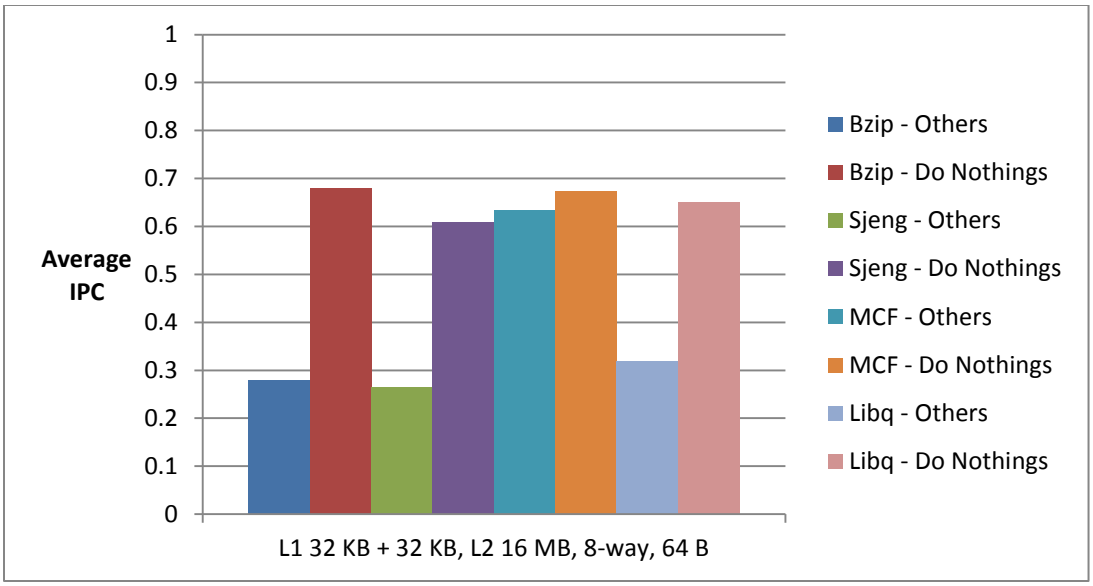

Figure 39. Average IPC for 16-Core Shared L2

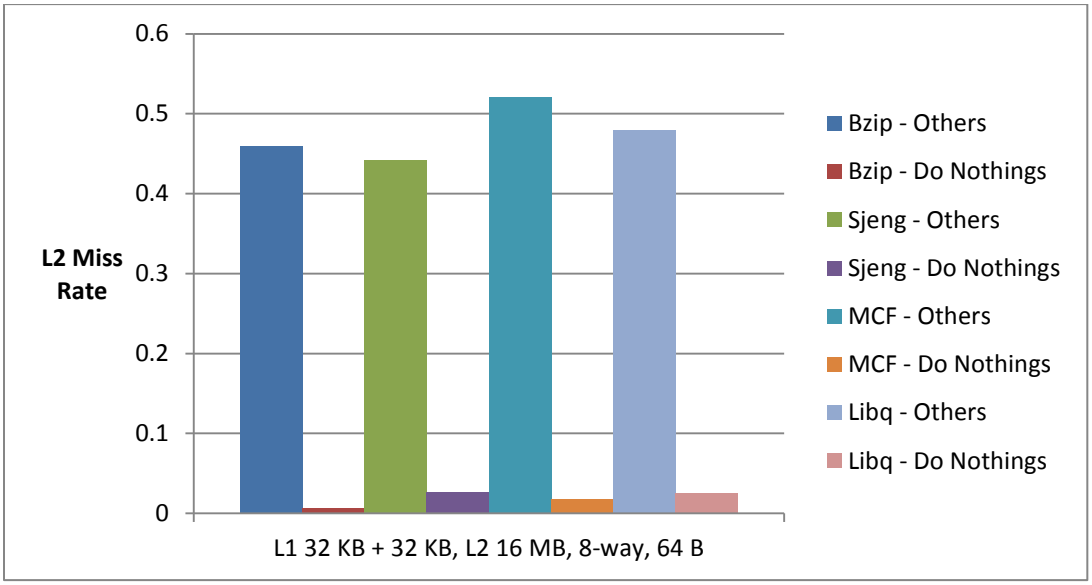

Figure 40. L2 Miss Rate for 16-Core Shared L2

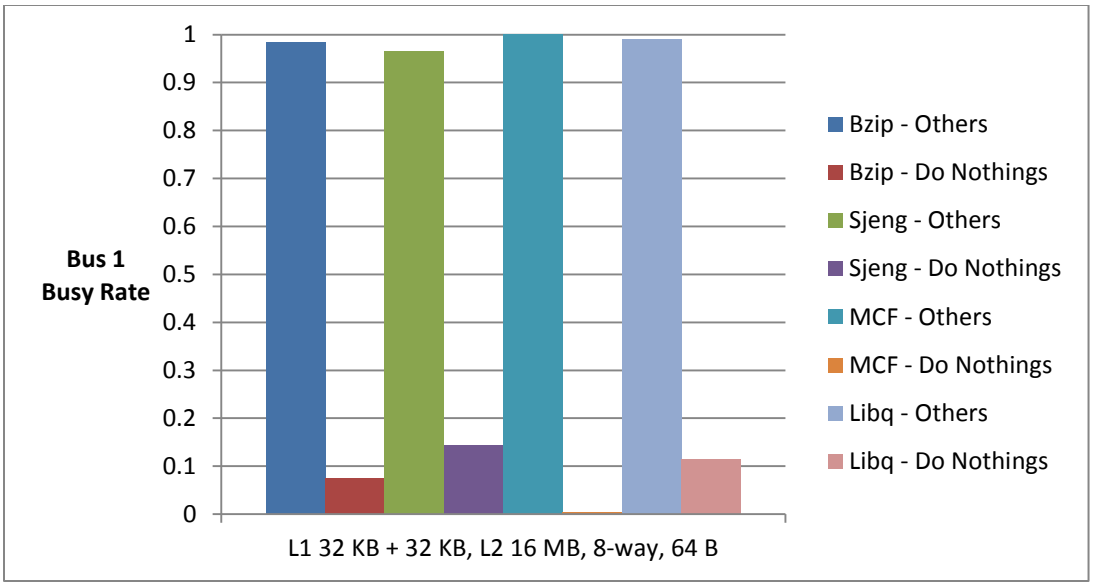

Figure 41. Bus 1 Busy Rate for 16-Core Shared L2

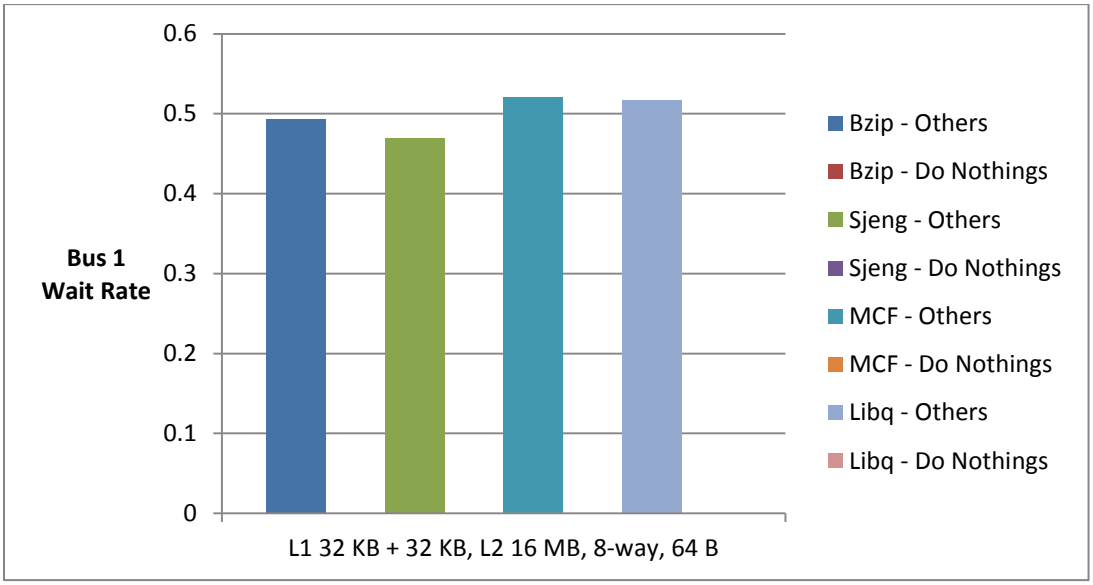

Figure 42. Bus 1 Wait Rate for 16-Core Shared L2

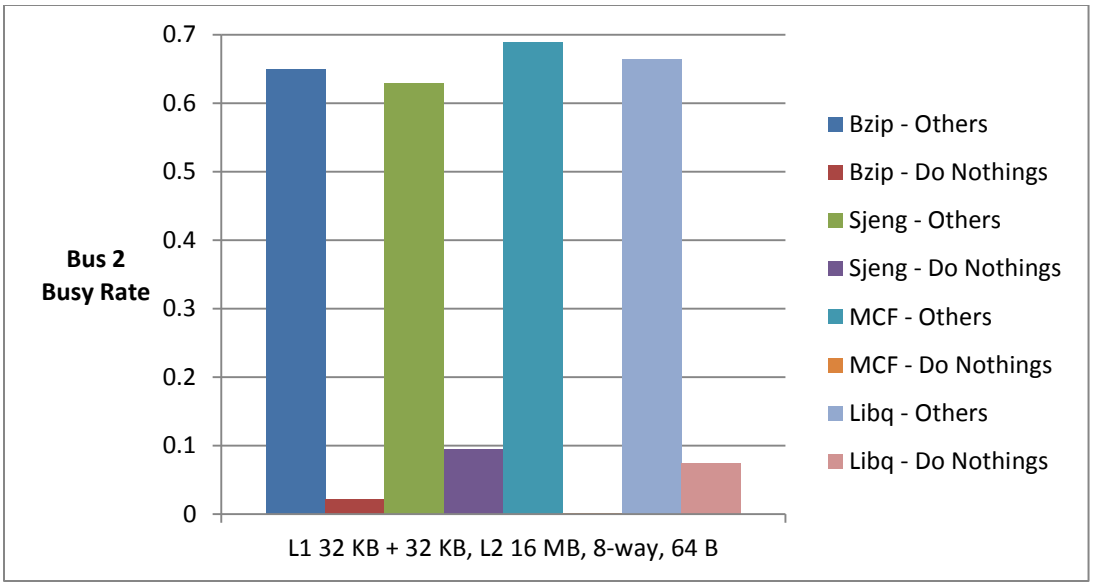

Figure 43. Bus 2 Busy Rate for 16-Core Shared L2

The bus 1 contention is virtually 100% hence it is not desirable to use a shared L2 architecture to handle 16 processors or more. The bus 2 contention is well over 60% and it is higher compared to the result from Octal-core Shared L2. Compared to the Octal-Core Shared L2 result, the L2 miss rate does not change. Chapter 4.5 will discuss the impact of using two processors using a shared L3 cache and we hope that the results are better than using a dual-core shared L2 architecture.

4.5 Dual-Core Shared L3

The Dual-Core Shared L3 architecture is shown in Figure 44. The recorded Average IPC, L2 Miss Rate, Bus 2 Busy Rate, Bus 2 Wait Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 45 – 48 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

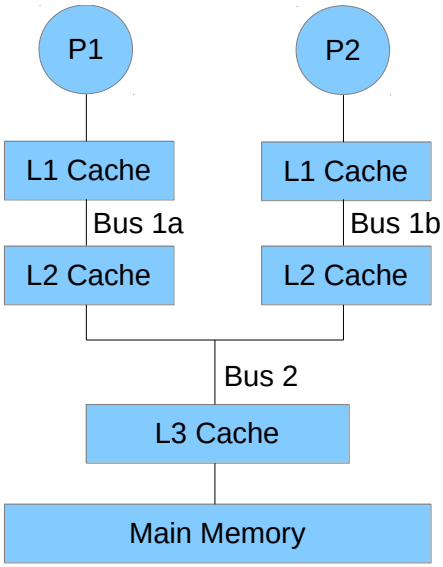

Figure 44. Dual-Core Shared L3 Architecture

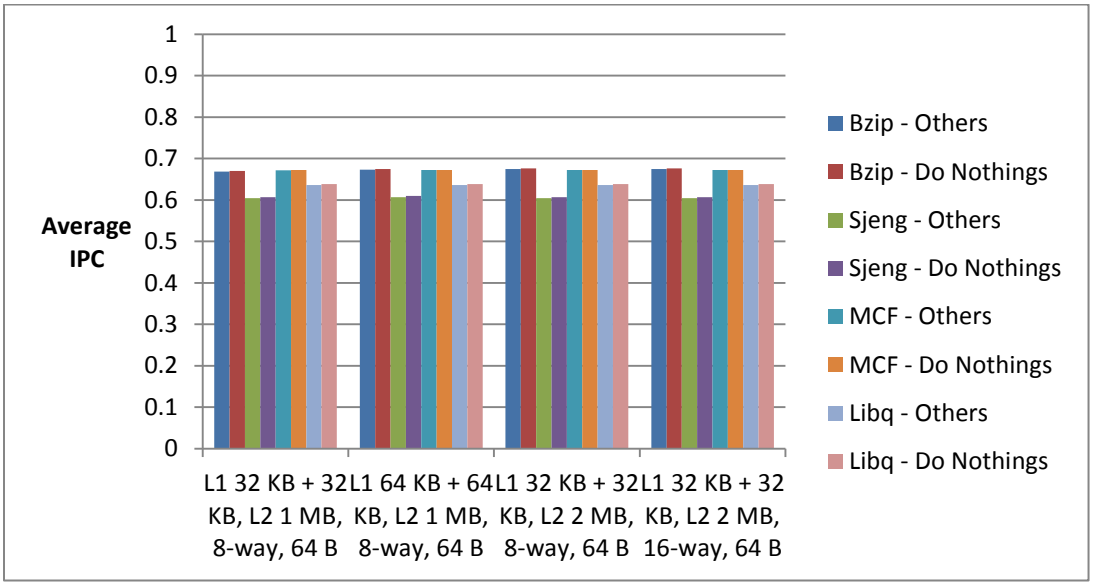

Figure 45. Average IPC for Dual-Core Shared L3

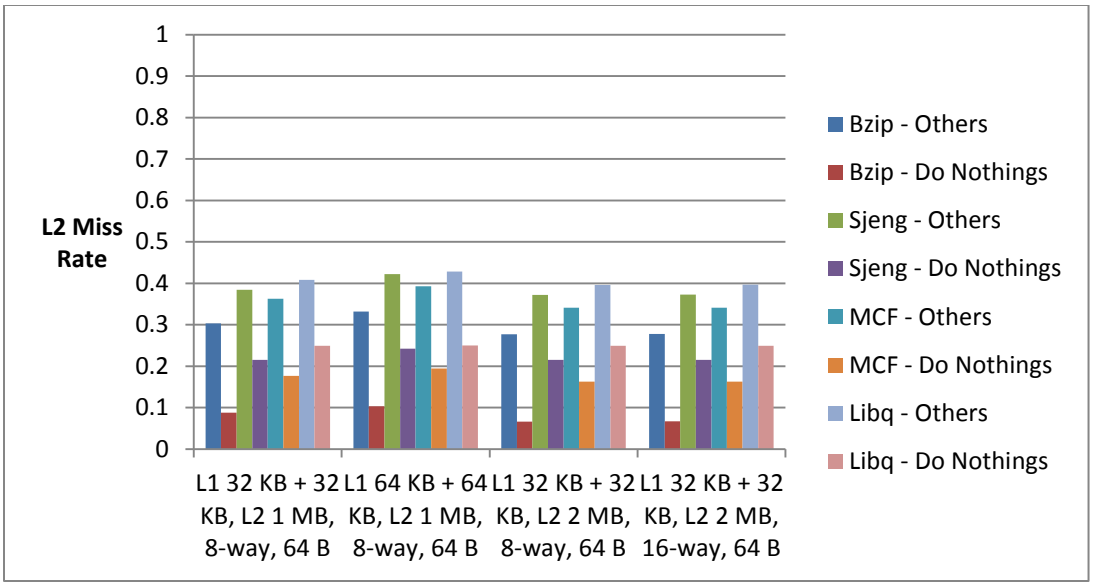

Figure 46. L2 Miss Rate for Dual-Core Shared L3

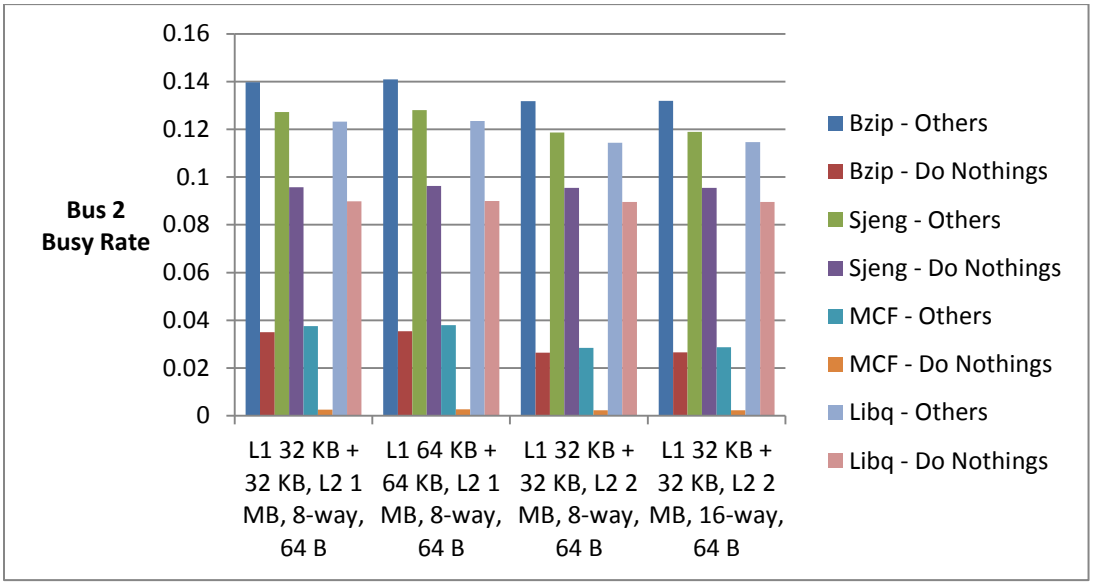

Figure 47. Bus 2 Busy Rate for Dual-Core Shared L3

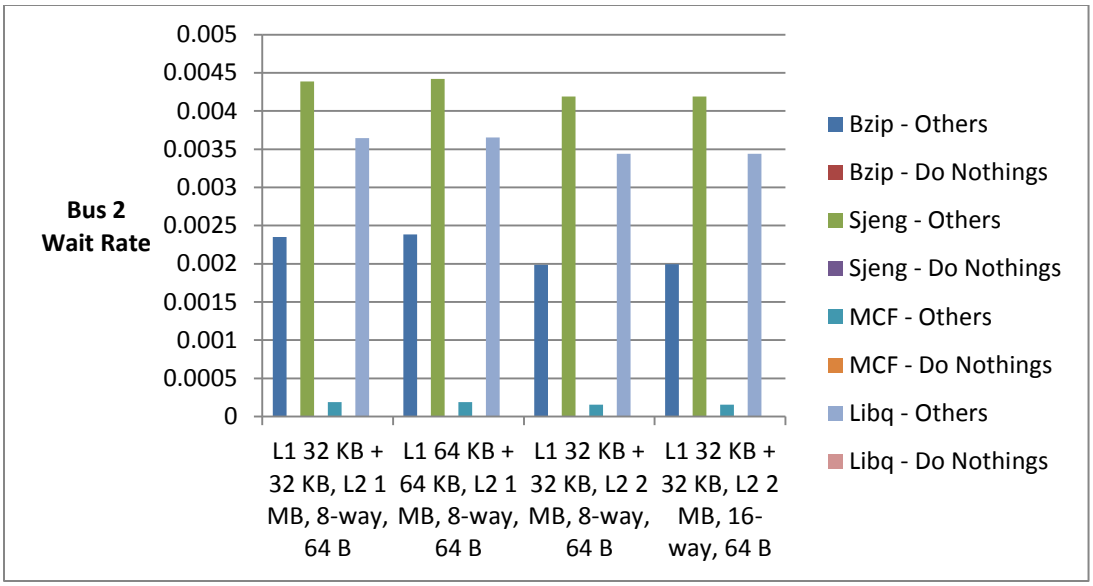

Figure 48. Bus 2 Wait Rate for Dual-Core Shared L3

As shown in Figure 45 - 48, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 2 busy rate, and bus 2 wait rate due to the fairly low bus contention. There is no significant improvement in performance between a dual-core shared L2 and L3 architecture. Chapter 4.6 will discuss the impact of using four processors using a shared L3 cache.

4.6 Quad-Core Shared L3

The Quad-Core Shared L3 architecture is shown in Figure 49. The recorded Average IPC, L2 Miss Rate, Bus 2 Busy Rate, Bus 2 Wait Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 50 – 53 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

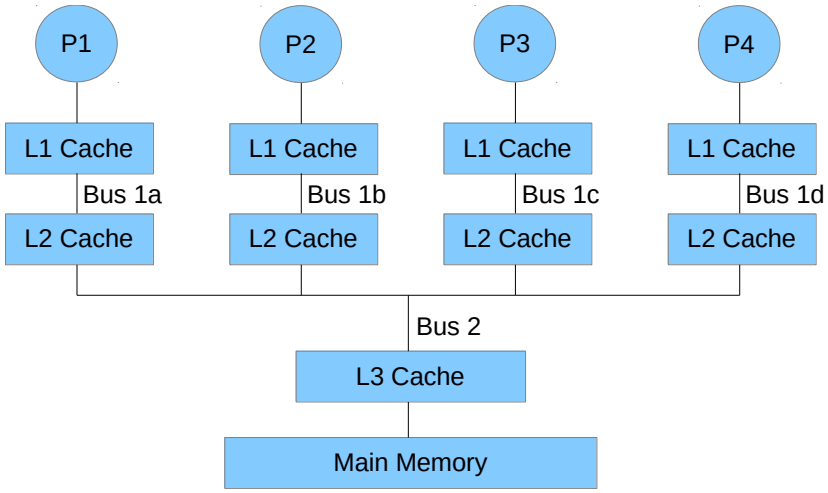

Figure 49. Quad-Core Shared L3 Architecture

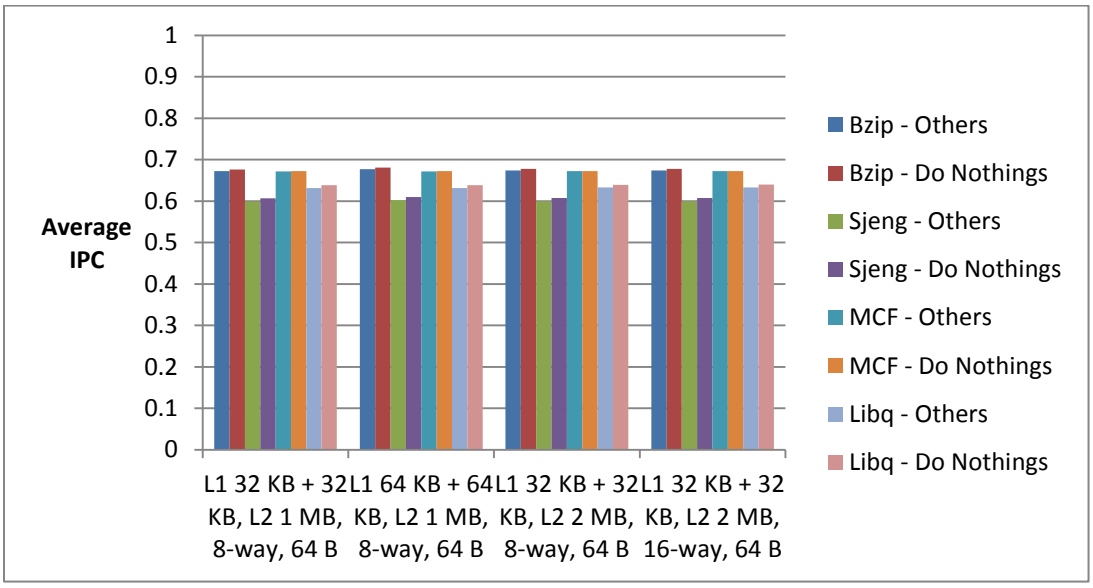

Figure 50. Average IPC for Quad-Core Shared L3

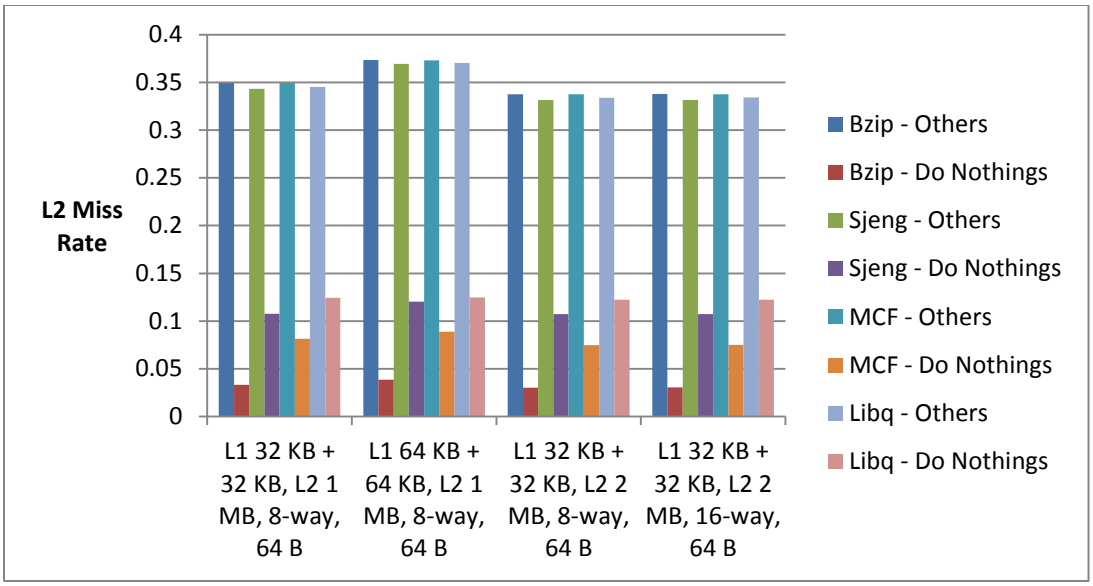

Figure 51. L2 Miss Rate for Quad-Core Shared L3

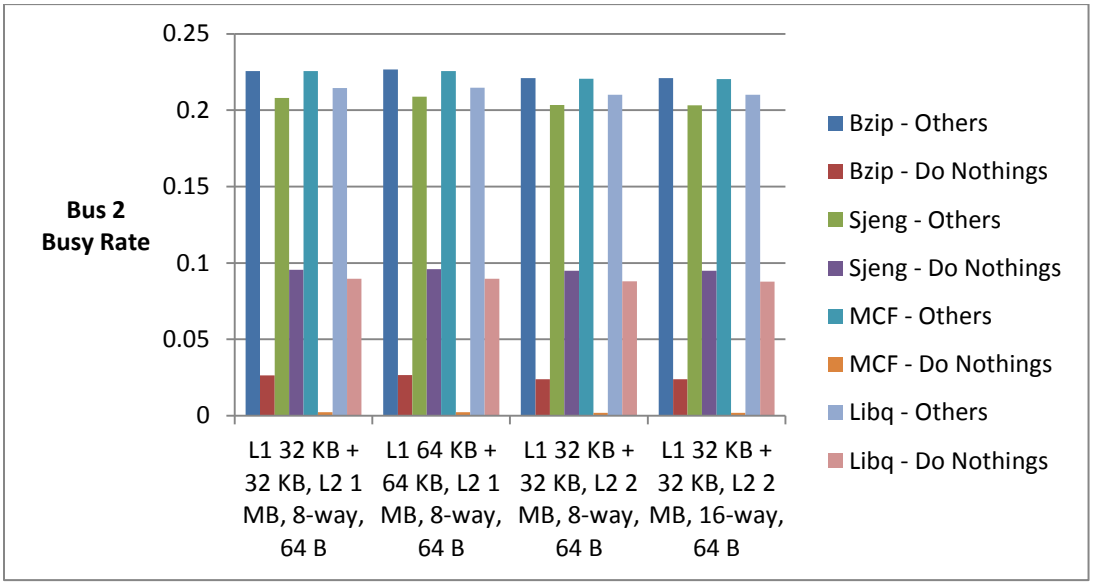

Figure 52. Bus 2 Busy Rate for Quad-Core Shared L3

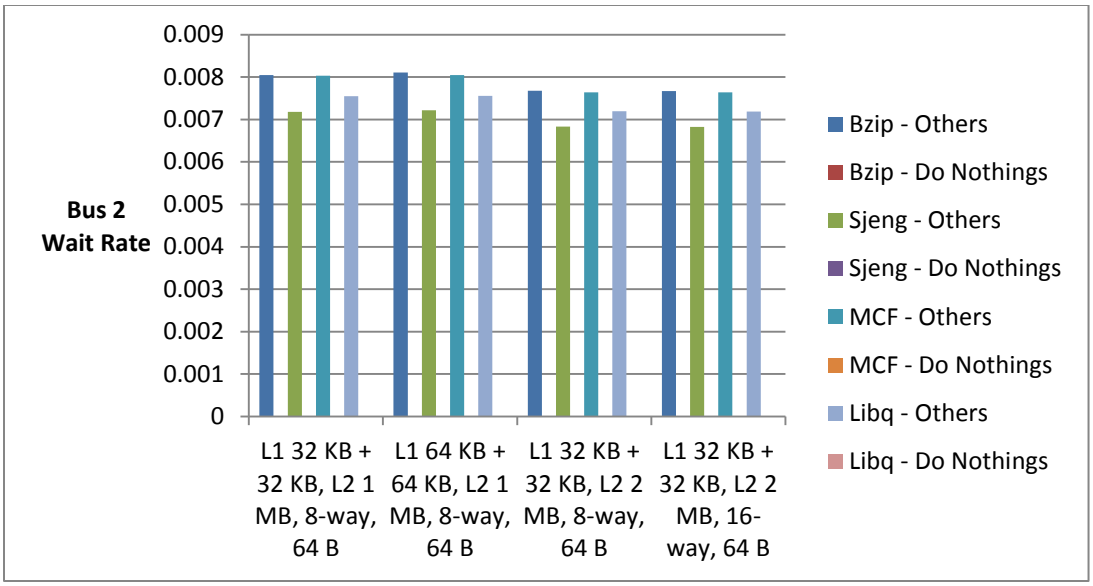

Figure 53. Bus 2 Wait Rate for Quad-Core Shared L3

As shown in Figure 50 - 53, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 2 busy rate, and bus 2 wait rate. Compared to the Quad-Core Shared L2 result, the L2 miss rate does not change that much. On average, the bus 2 busy rate for quad-core shared L3 is lower than quad-core shared L2. We are hoping to see more of this phenomenon in Chapter 4.7. Chapter 4.7 will discuss the impact of using eight processors using a shared L3 cache.

4.7 Octal-Core Shared L3

The Octal-Core Shared L3 architecture is shown in Figure 54. The recorded Average IPC, L2 Miss Rate, Bus 2 Busy Rate, Bus 2 Wait Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 55 – 58 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

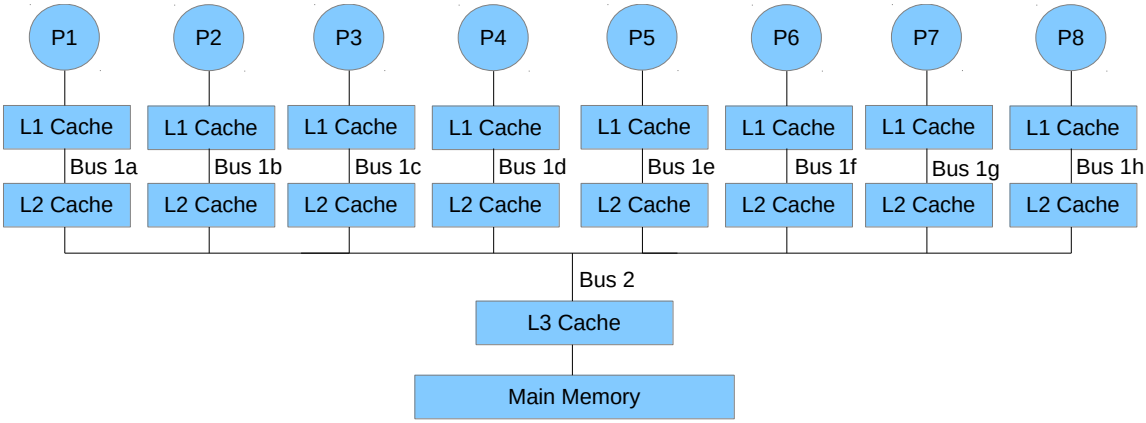

Figure 54. Octal-Core Shared L3 Architecture

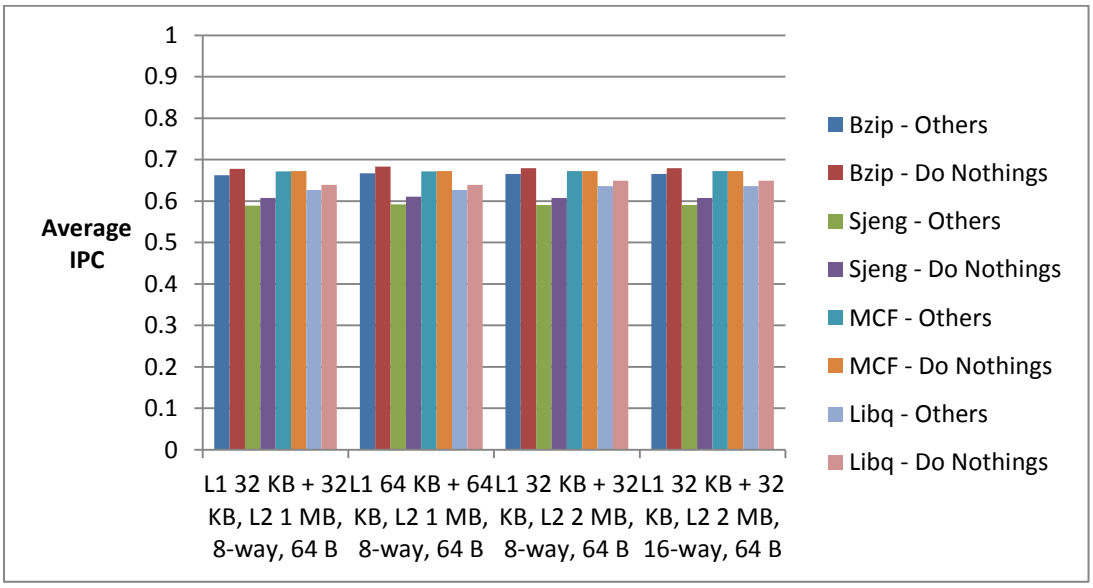

Figure 55. Average IPC for Octal-Core Shared L3

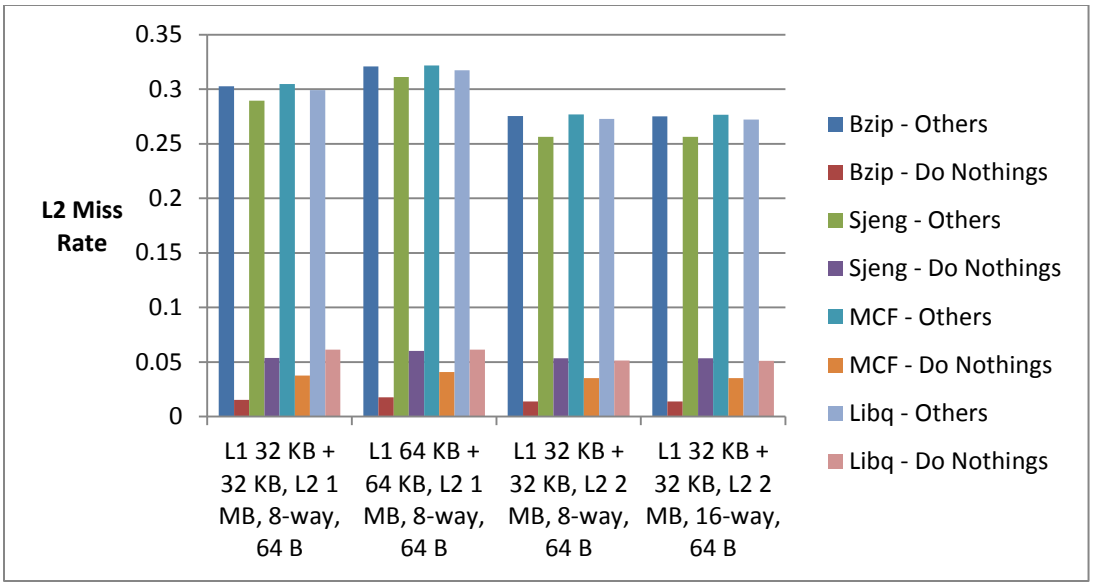

Figure 56. L2 Miss Rate for Octal-Core Shared L3

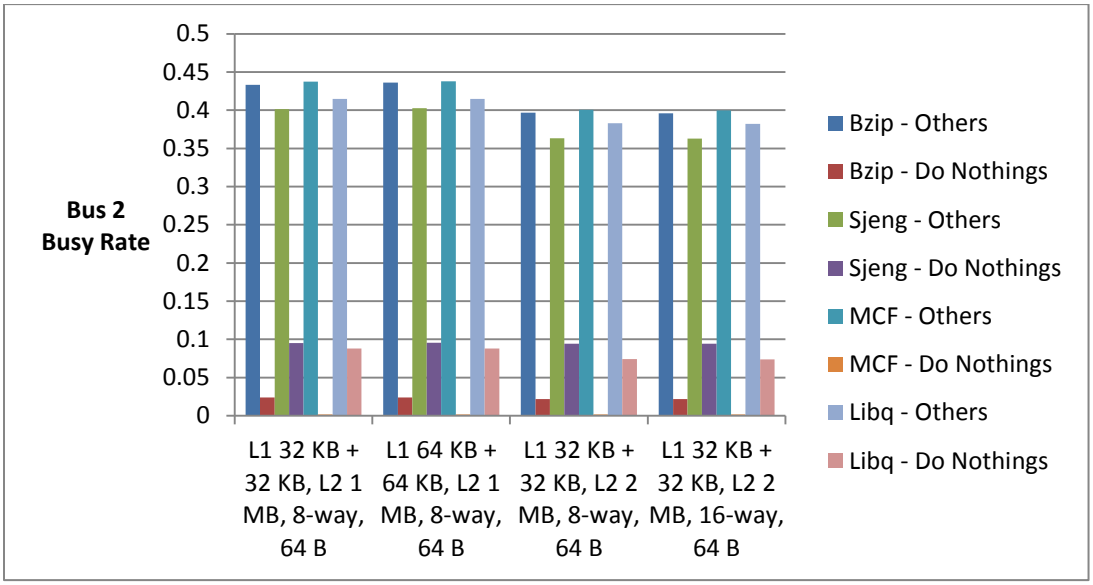

Figure 57. Bus 2 Busy Rate for Octal-Core Shared L3

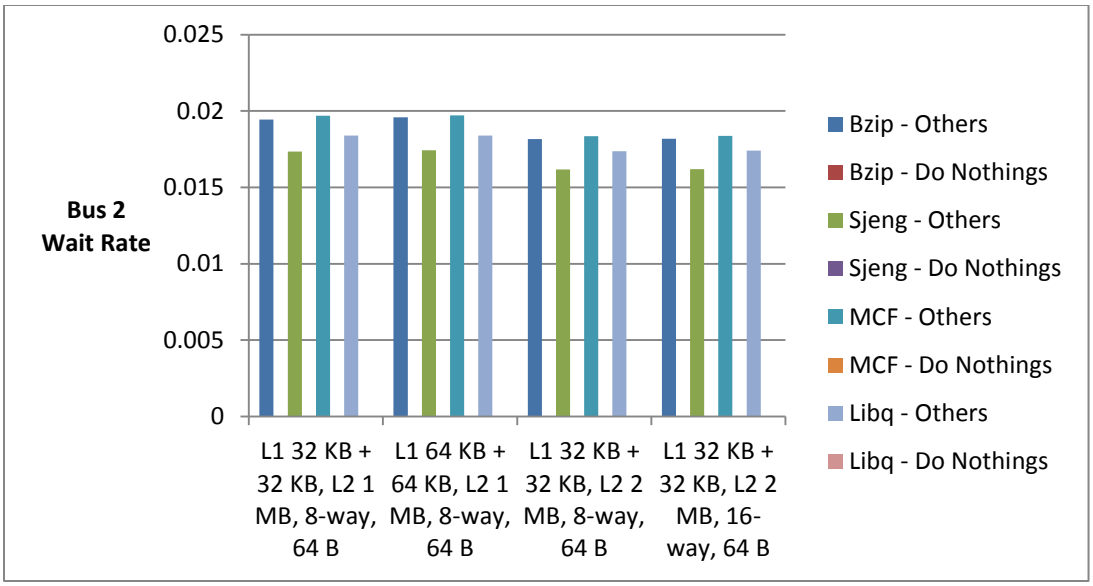

Figure 58. Bus 2 Wait Rate for Octal-Core Shared L3

As shown in Figure 55 - 58, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 2 busy rate, and bus 2 wait rate. Compared to the Octal-Core Shared L2 result, the L2 miss rate does not change that much. On average, the bus 2 busy rate for octal-core shared L3 is lower than octal-core shared L2. Chapter 4.8 will discuss the impact of using sixteen processors using a shared L3 cache.

4.8 16-Core Shared L3

The 16-Core Shared L3 architecture is shown in Figure 59. The recorded Average IPC, L2 Miss Rate, Bus 2 Busy Rate, Bus 2 Wait Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 60 – 63 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

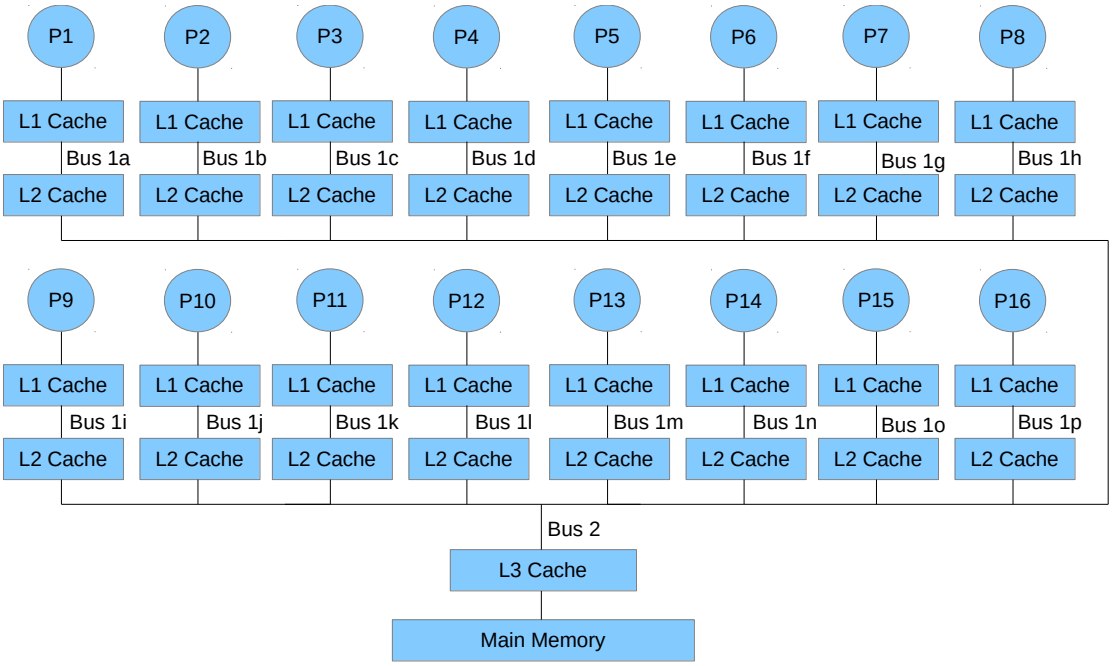

Figure 59. 16-Core Shared L3 Architecture

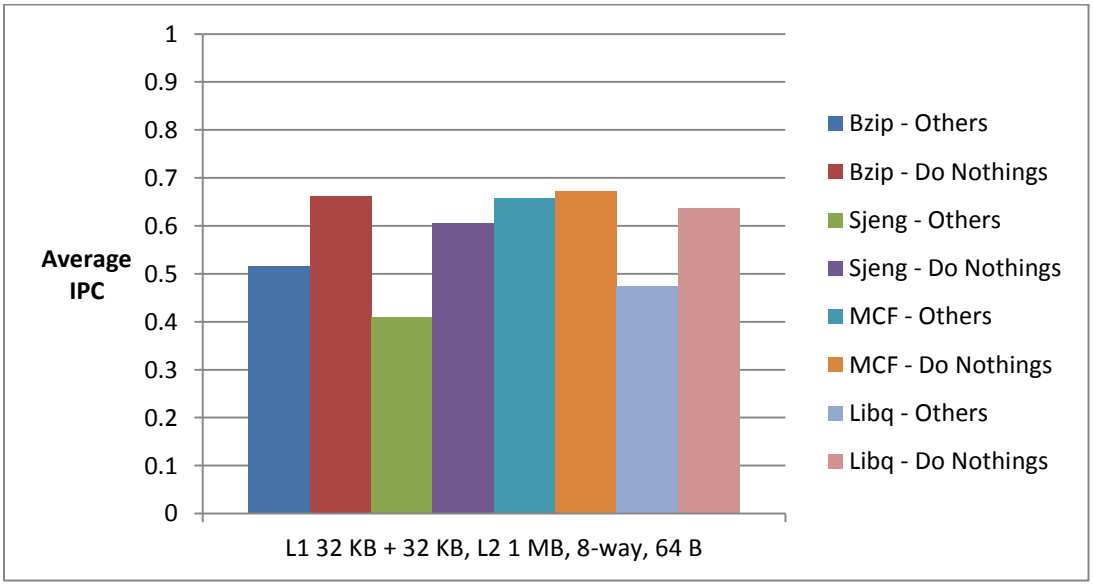

Figure 60. Average IPC for 16-Core Shared L3

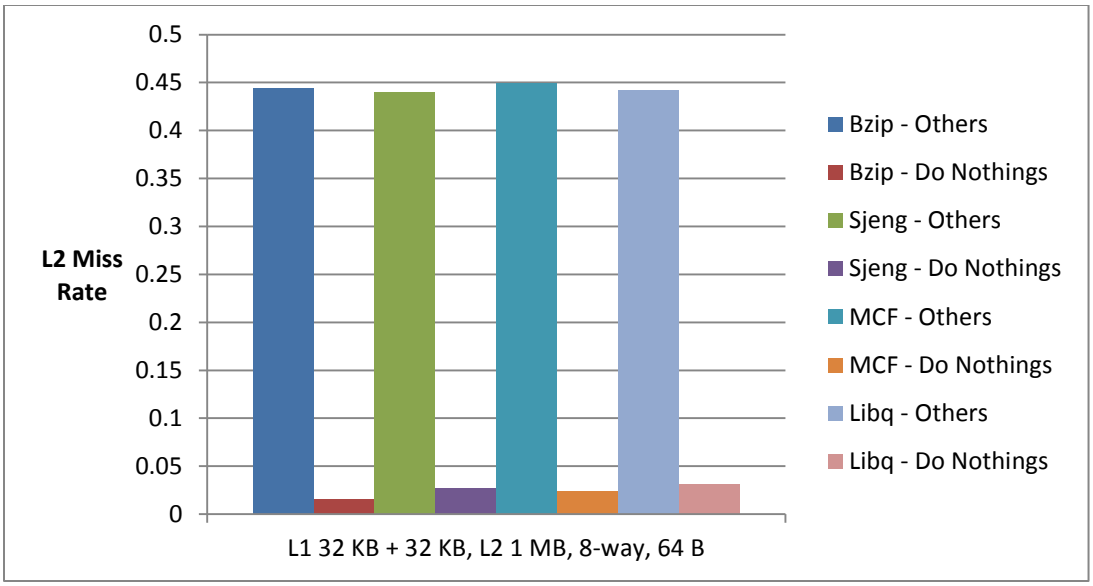

Figure 61. L2 Miss Rate for 16-Core Shared L3

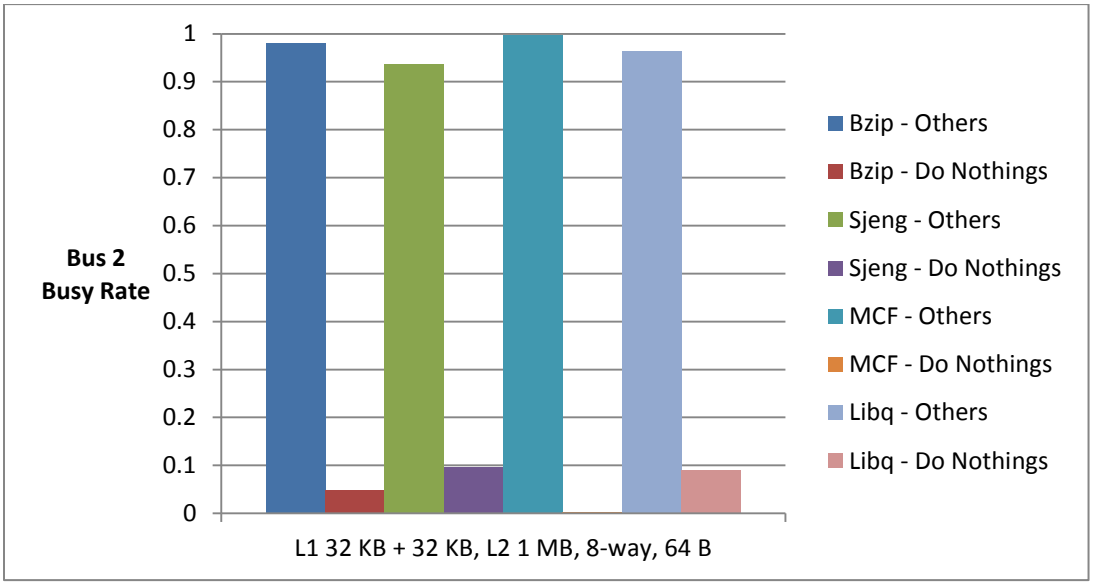

Figure 62. Bus 2 Busy Rate for 16-Core Shared L3

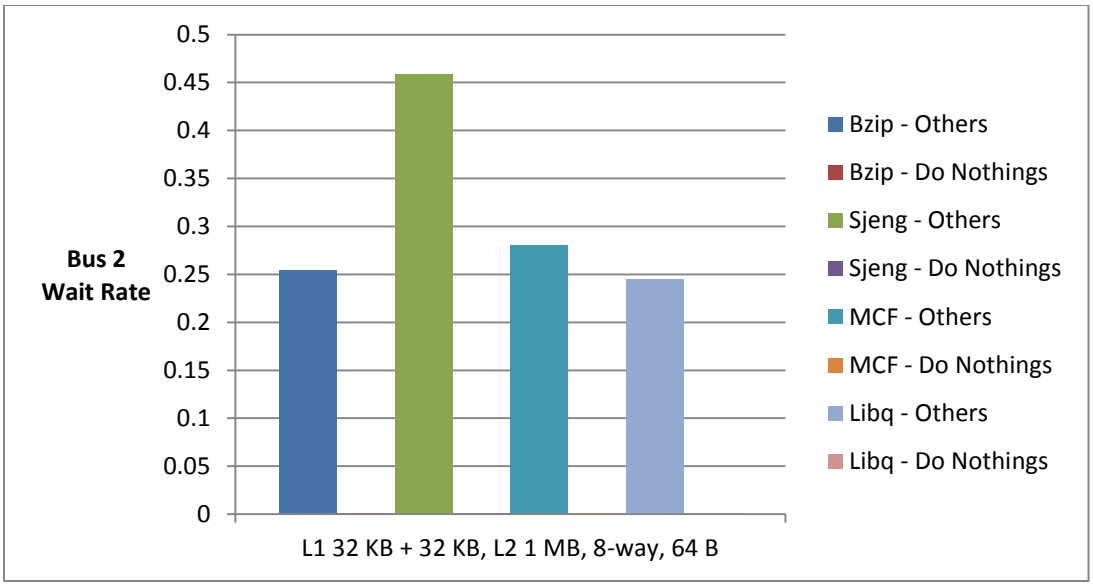

Figure 63. Bus 2 Wait Rate for 16-Core Shared L3

Compared to the Octal-Core Shared L2 result, the L2 miss rate does not change that much. The bus 2 busy rate is really close to 100% hence it is not desirable to use a shared L3 architecture to handle 16 processors or more. Chapter 4.9 will discuss the impact of using four processors using a hierarchical architecture and we hope that the results are better than using a quad-core shared L2 and L3 architecture.

4.9 Quad-Core Hierarchy

The Quad-Core Hierarchy architecture is shown in Figure 64. The recorded Average IPC, L2 Miss Rate, Bus 1 Busy Rate, Bus 2 Busy Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 65 – 68 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

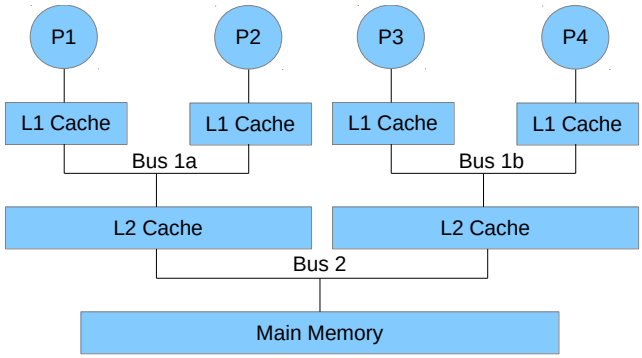

Figure 64. Quad-Core Hierarchy Architecture

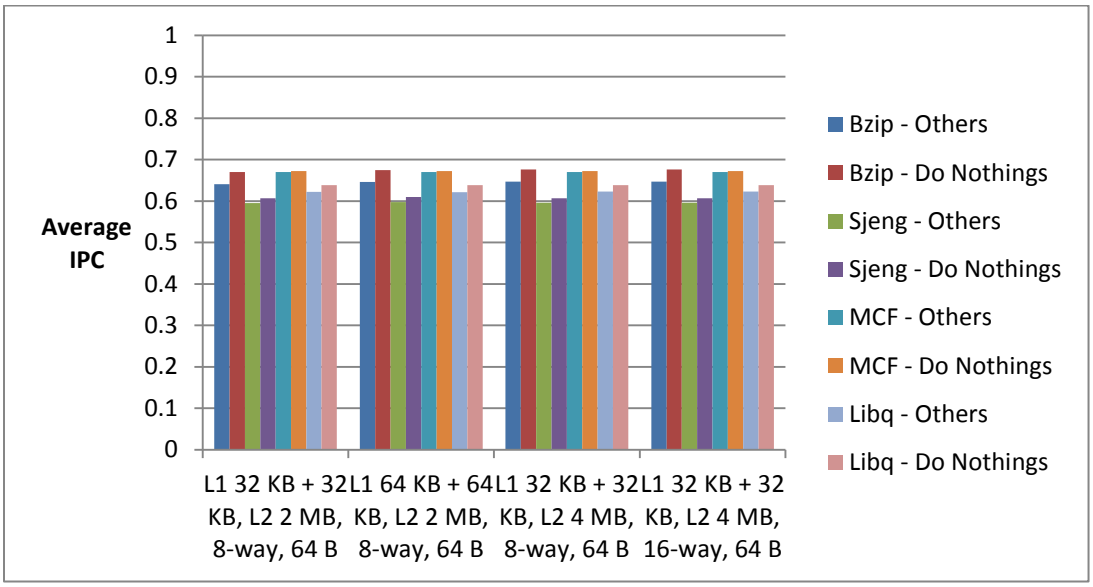

Figure 65. Average IPC for Quad-Core Hierarchy

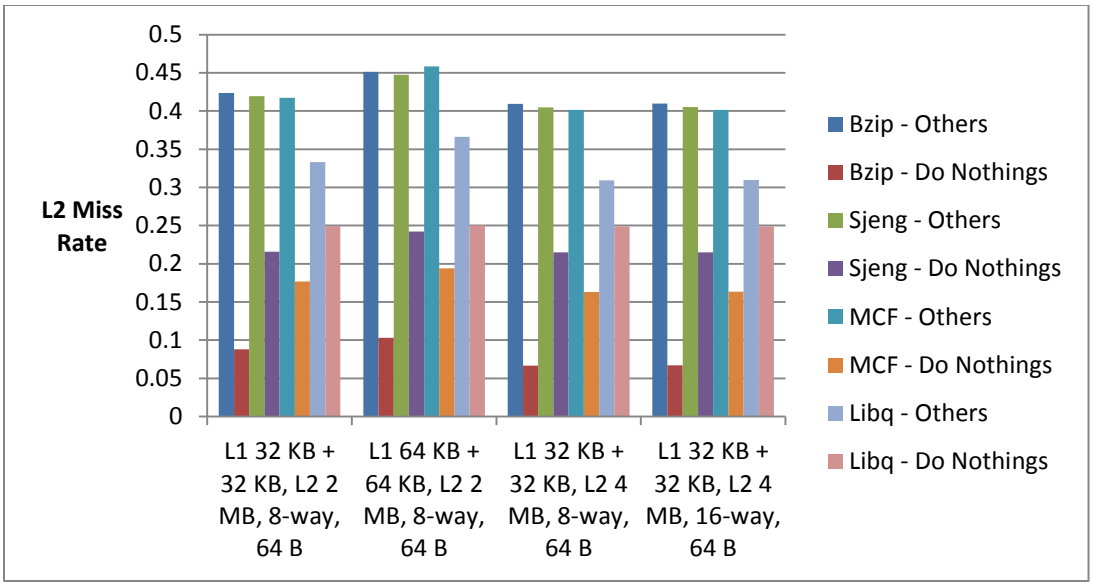

Figure 66. L2 Miss Rate for Quad-Core Hierarchy

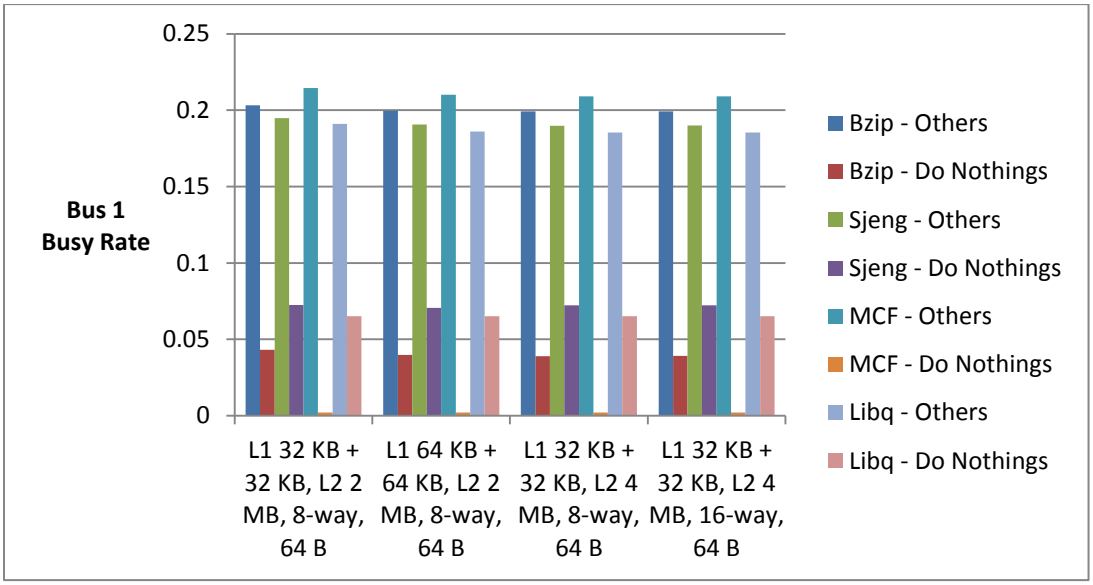

Figure 67. Bus 1 Busy Rate for Quad-Core Hierarchy

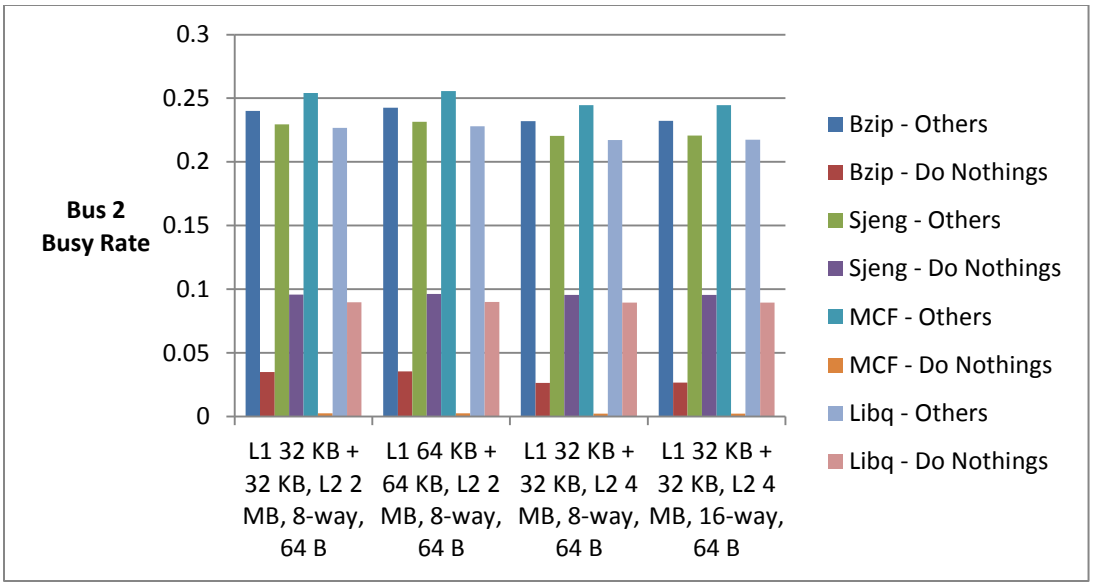

Figure 68. Bus 2 Busy Rate for Quad-Core Shared L3

As shown in Figure 65 - 68, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 2 busy rate, and bus 2 wait rate. On average, the IPC performance for Hierarchy falls between the quad-core shared L2 and L3 but the bus 2 busy rate is not significantly better than a shared L2 architecture. The bus 1 busy rate for hierarchy is lower than a quad-core shared L2 architecture. Chapter 4.10 will discuss the impact of using eights processors using a hierarchical architecture.

4.10 Octal-Core Hierarchy

The Octal-Core Hierarchy architecture is shown in Figure 69. The recorded Average IPC, L2 Miss Rate, Bus 1 Busy Rate, Bus 2 Busy Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 70 – 73 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

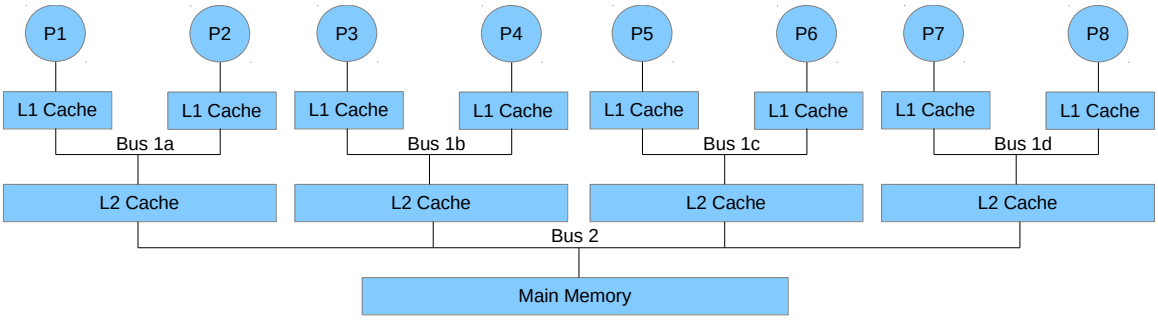

Figure 69. Octal-Core Hierarchy Architecture

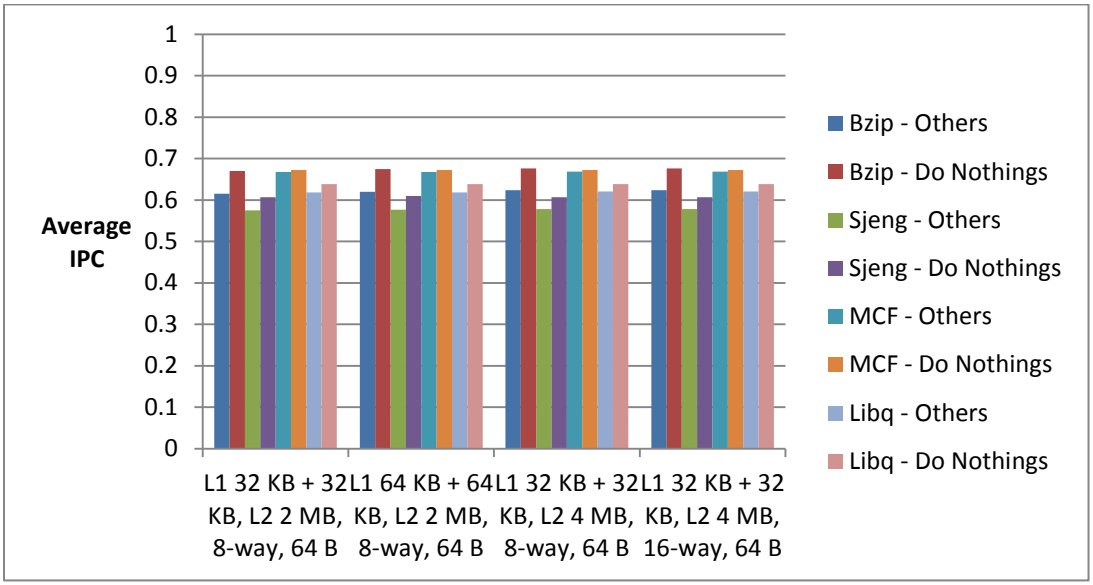

Figure 70. Average IPC for Octal-Core Hierarchy

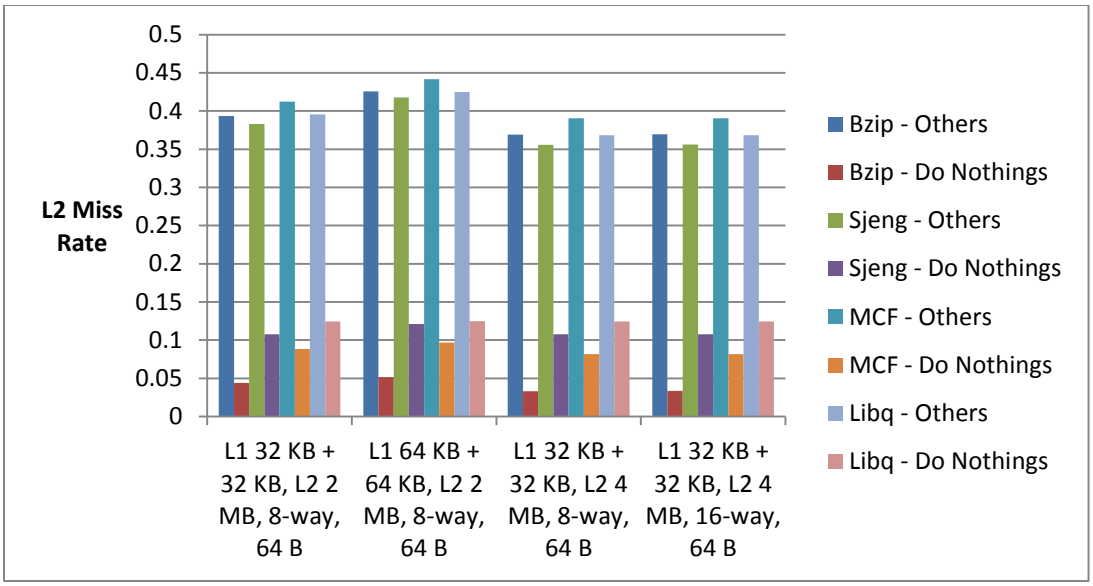

Figure 71. L2 Miss Rate for Octal-Core Hierarchy

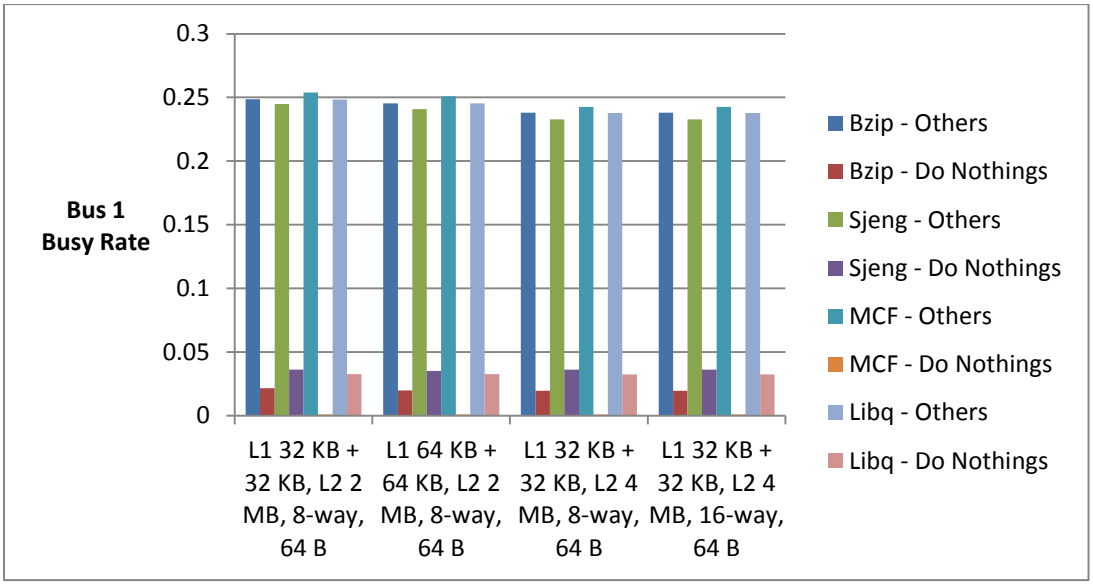

Figure 72. Bus 1 Busy Rate for Octal-Core Hierarchy

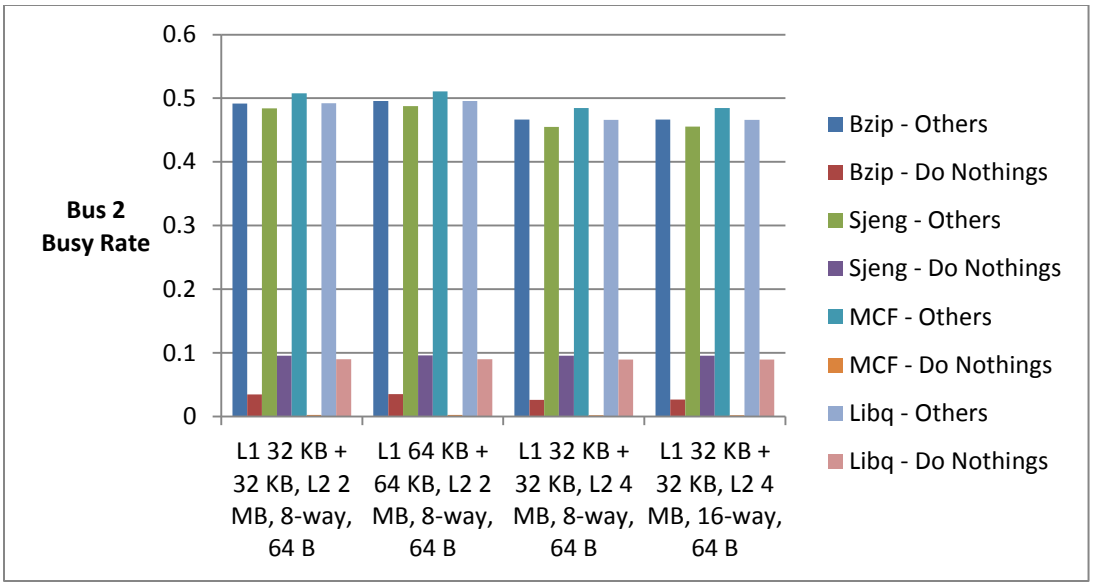

Figure 73. Bus 2 Busy Rate for Octal-Core Hierarchy

As shown in Figure 71 - 73, doubling the size of L1 or L2 cache does not significantly increase the average IPC (Instructions Per Cycle), and reduce the L2 miss rate, bus 2 busy rate, and bus 2 wait rate. On average, the IPC performance for Hierarchy falls between the octal-core shared L2 and L3 but the bus 2 busy rate is not significantly better than a shared L2 architecture. The bus 1 busy rate for hierarchy is lower than an octal-core shared L2 architecture. Chapter 4.11 will discuss the impact of using sixteen processors using a hierarchical architecture.

4.11 16-Core Hierarchy

The 16-Core Hierarchy architecture is shown in Figure 74. The recorded Average IPC, L2 Miss Rate, Bus 1 Busy Rate, Bus 2 Busy Rate for mixed program (Bzip – Others, Sjeng – Others, MCF – Others and Libq – Others) and single program (Bzip – Do Nothing, Sjeng – Do Nothing, MCF – Do Nothing and Libq – Do Nothing) shown in Figure 75 – 78 are determined over the number of clock cycles for the named benchmark run to 1 billion instructions.

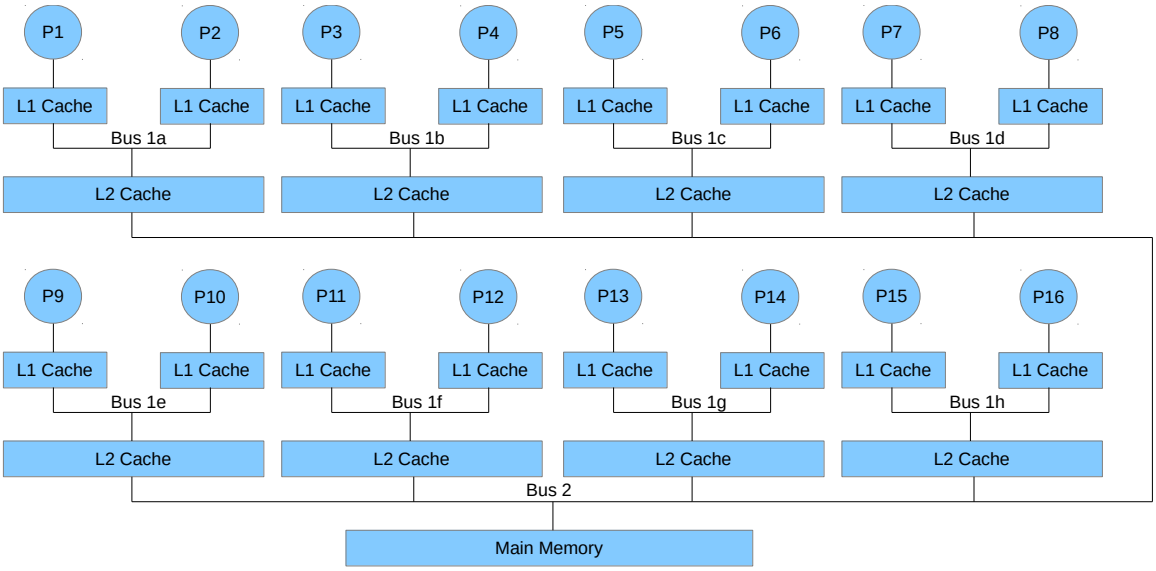

Figure 74. 16-Core Hierarchy Architecture

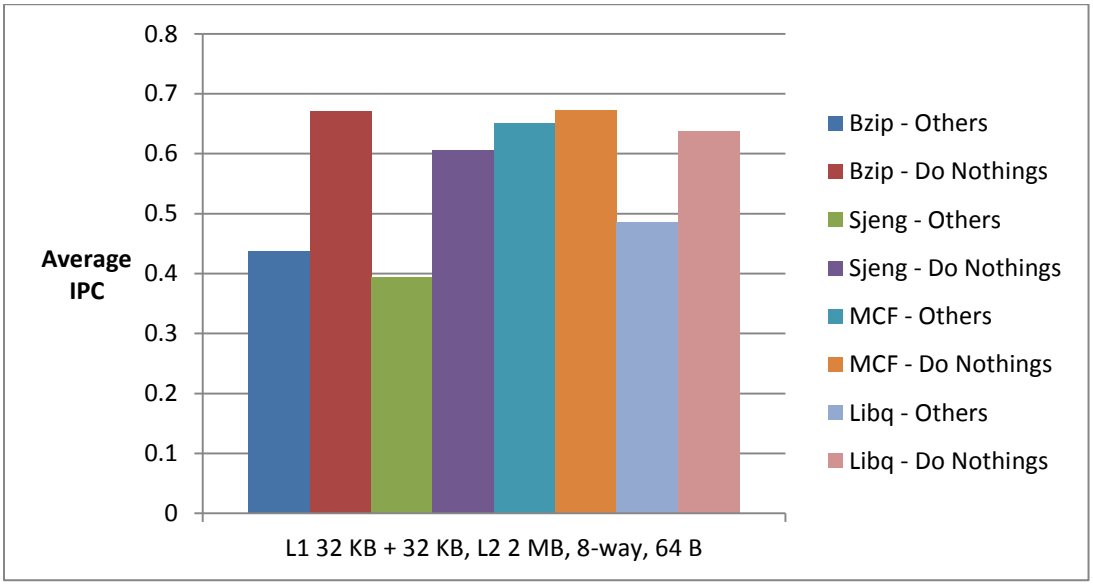

Figure 75. Average IPC for 16-Core Hierarchy

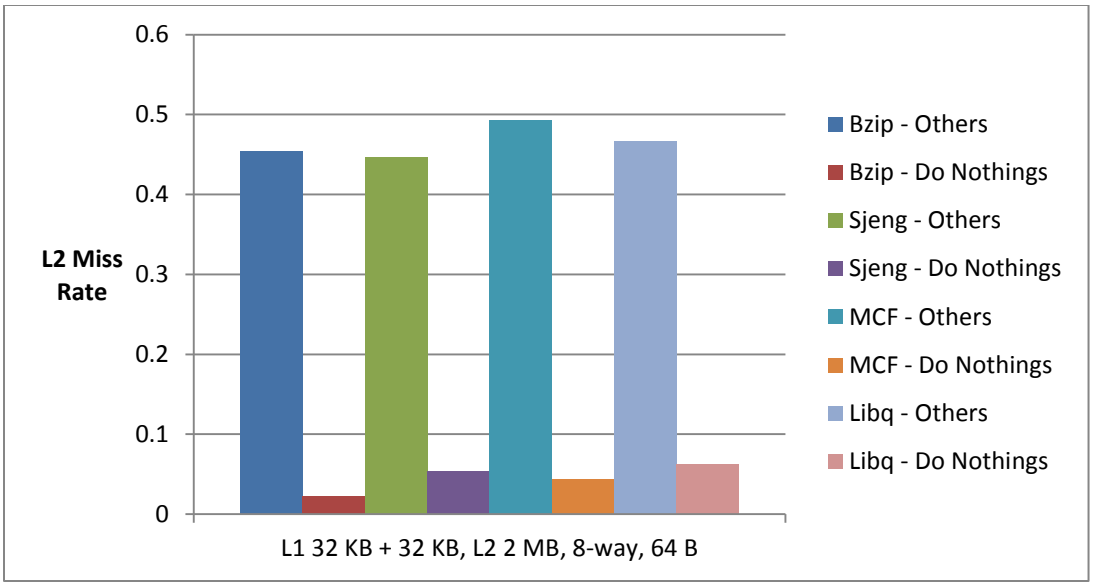

Figure 76. L2 Miss Rate for 16-Core Hierarchy

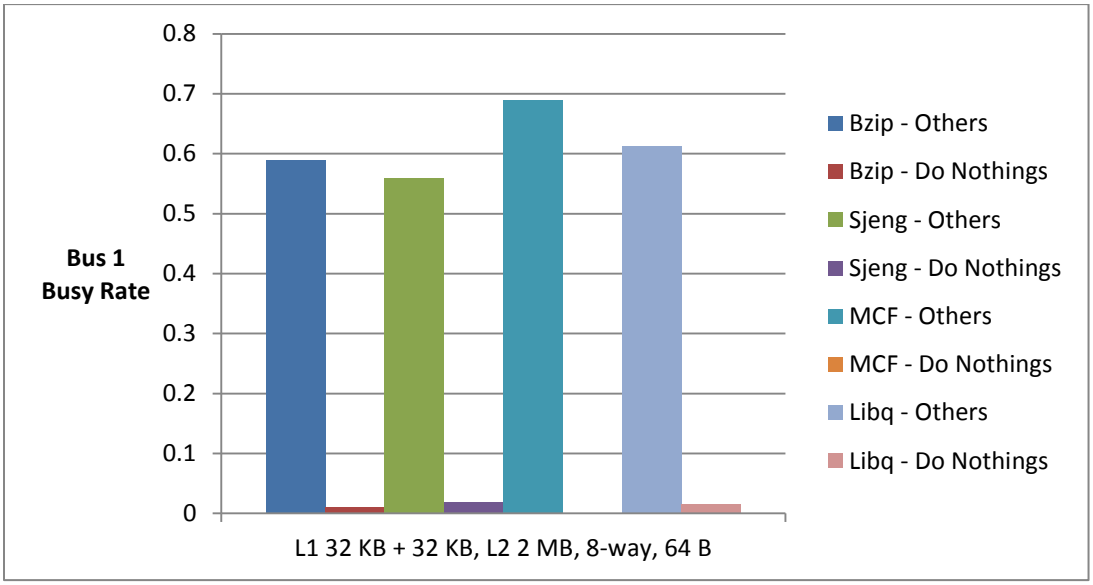

Figure 77. Bus 1 Busy Rate for 16-Core Hierarchy

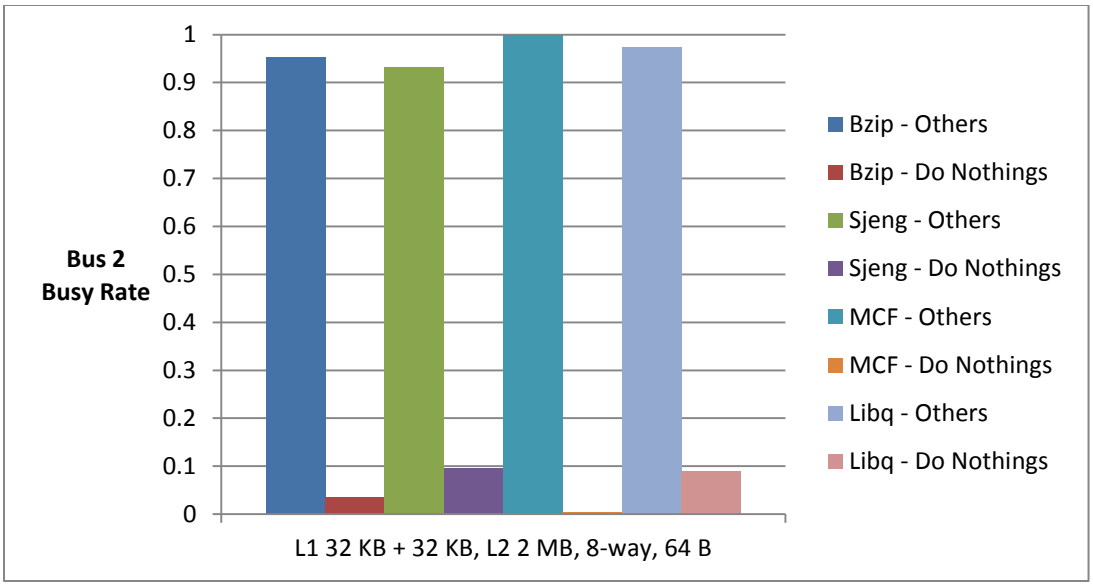

Figure 78. Bus 2 Busy Rate for 16-Core Hierarchy

Compared to the Octal-Core Hierarchical result, the L2 miss rate does not change that much due to the fairly high bus contention. The bus 2 busy rate is really close to 100% hence it is not desirable to use a Hierarchical architecture to handle 16 processors or more. Chapter 4.12 will summarize our findings.

4.12 Performance Comparison

Using the data in Chapter 4.1 to 4.11, we can plot the performance for single, dual, quad, octal and 16 core shown in Figure 79 - 84. Figure 79 shows the average IPC performance per processor running all benchmarks. Figure 80 shows the average IPC performance for all processors on a chip running all benchmarks. Figure 81 shows the L2 miss rate for Shared L2, Shared L3 and Hierarchical architecture for mixed program. Figure 82 shows the performance of a Shared L2 architecture vs number of processors. Figure 83 shows the performance of a Shared L3 architecture vs number of processors. Figure 84 shows the average L2 miss rate vs number of

processors for single program using a shared L2 cache. Figure 85 shows the performance loss vs number of processors.

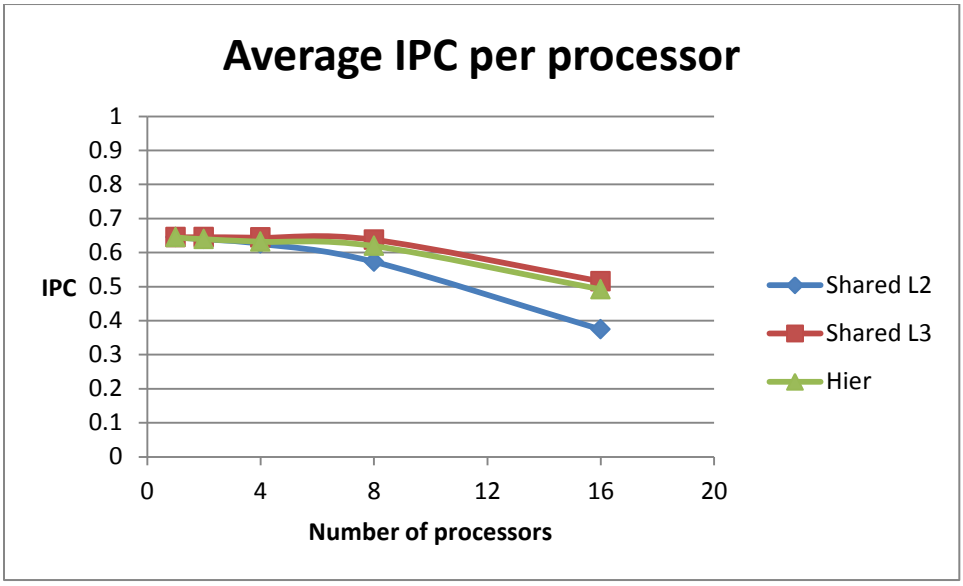

Figure 79. Average IPC performance per processor running all benchmarks

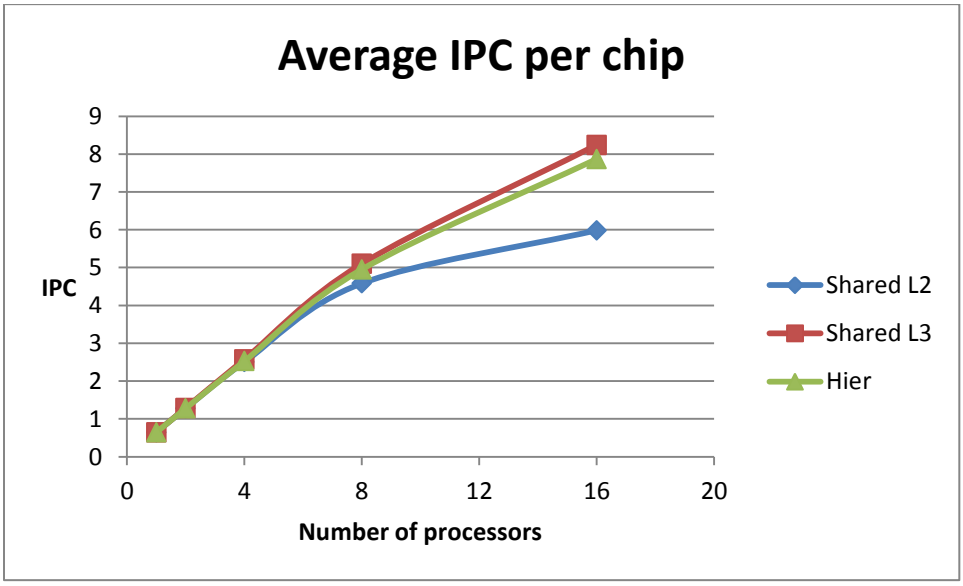

Figure 80. Average IPC performance for all processors on a chip running all benchmarks

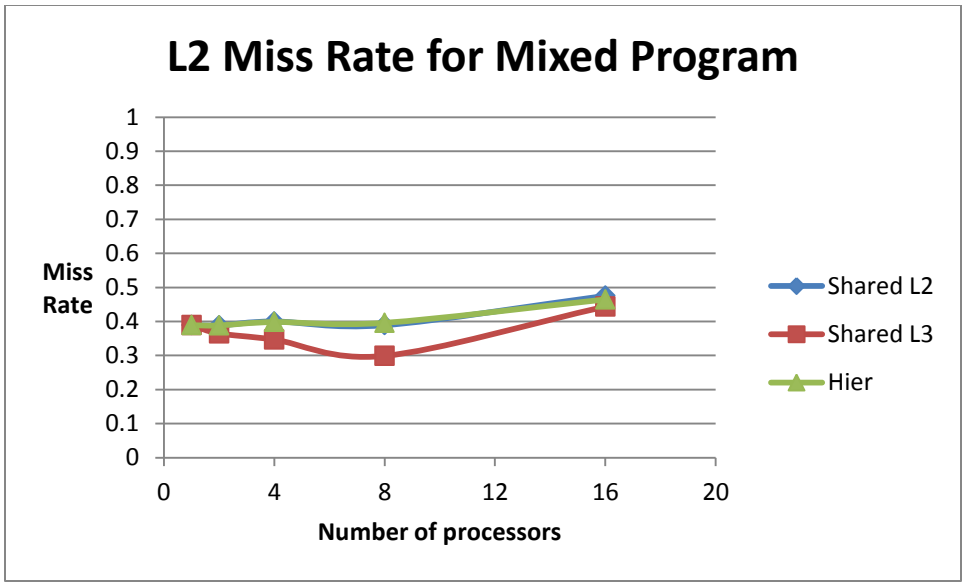

Figure 81. L2 miss rate for Shared L2, Shared L3, and Hierarchical cache architectures

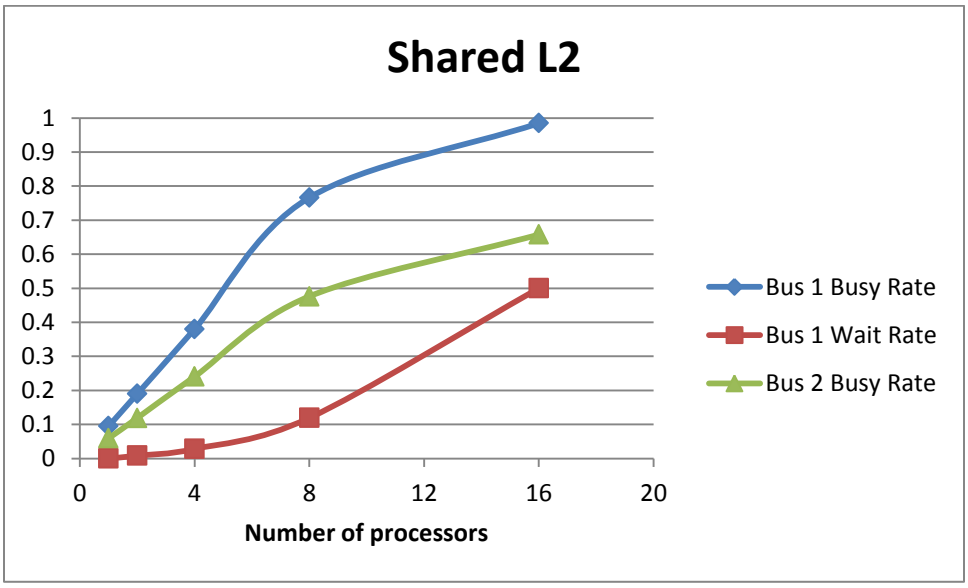

Figure 82. Shared L2 Performance

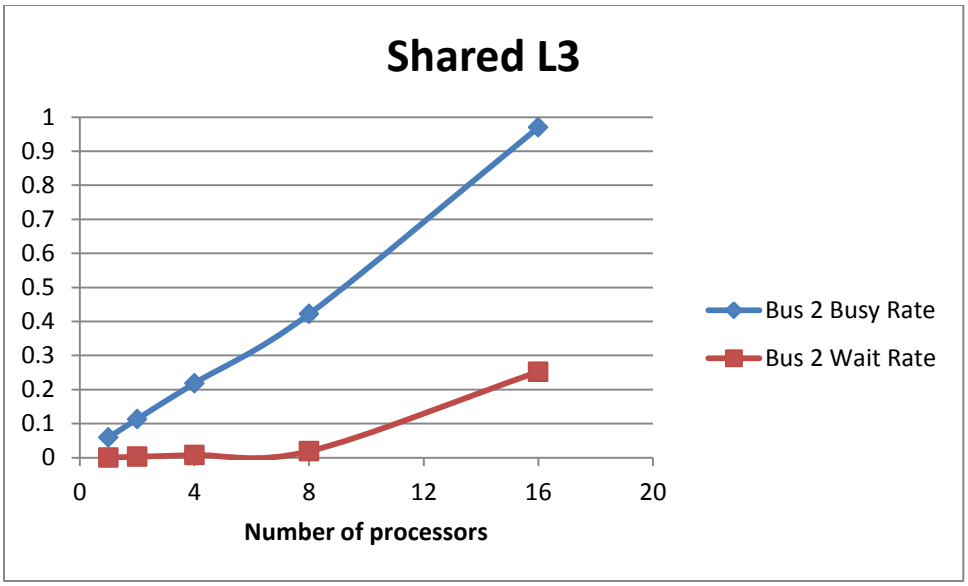

Figure 83. Shared L3 Performance

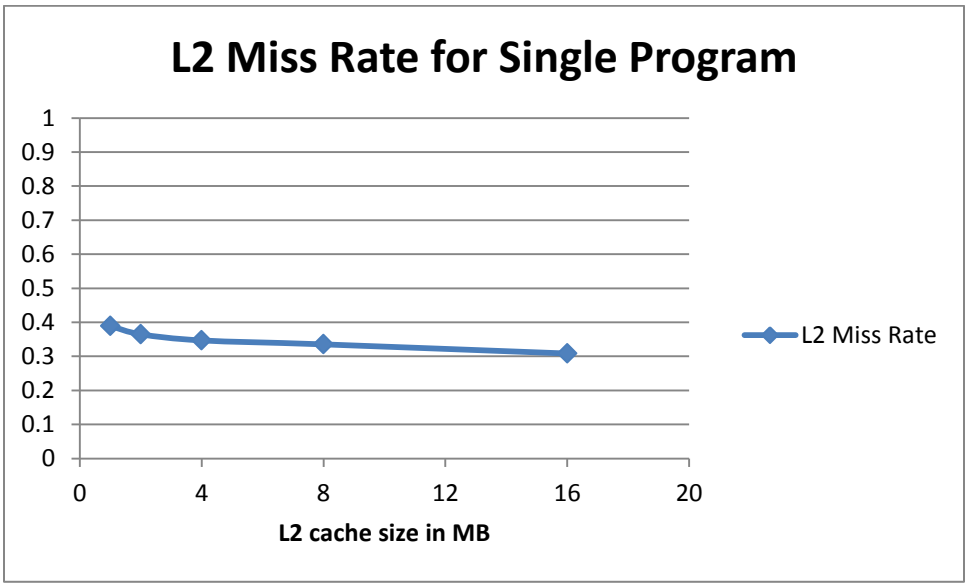

Figure 84. Average L2 miss rate vs L2 cache size for single program for Shared L2 cache

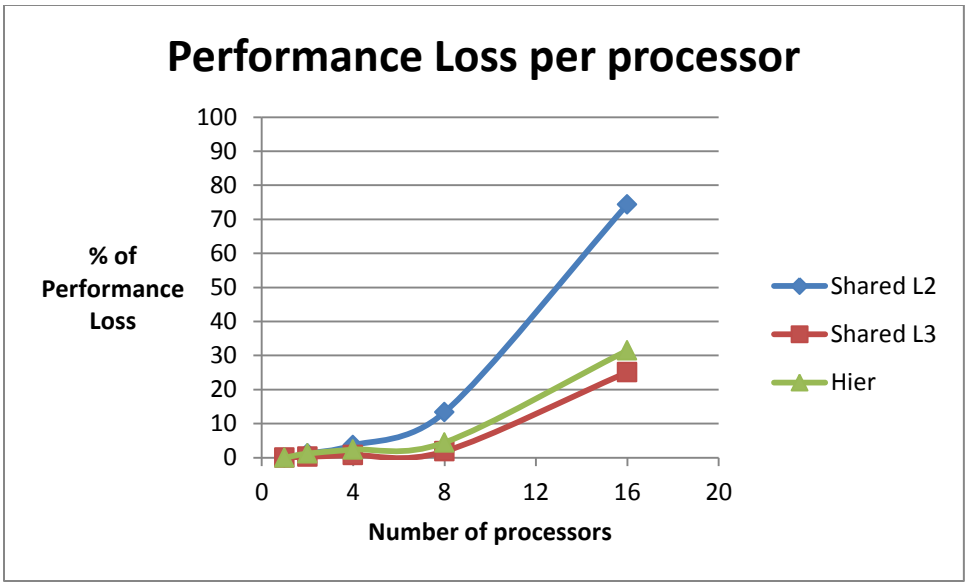

Figure 85. Performance Loss per processor vs Number of Processors

As shown in Figure 80, the rate of which Shared L2 declines is faster than Shared L3 and Hierarchical, hence sharing a L2 cache may not be the best way to handle a large number of processors. In Figure 81, one would expect that we would double the IPC as we double the number of processors; but as the number of processors increases we get less than we hope for. In Figure 82, one can see it is not desirable to use a shared L2 architecture to handle 16 processors or more due to the very high bus contention. In Figure 83, it is also not desirable to use a shared L3 architecture to handle 16 processors or more due to the very high bus contention but a processor waits about half of the time to get requests granted. Figure 84 emphasizes our findings that increasing the size of shared L2 cache does not solve the high bus contention problem. A more exotic approach needs to be explored and researched in order to reduce bus contention to handle 16 processors or more. Figure 85 shows that we have a bigger loss of performance as we increase the number of processors.

CHAPTER V

FUTURE WORK

For future work, we are interested in doing a cache performance evaluation for ring topology and multibank cache to compare with the shared L2 and L3 cache, and hierarchical cache architecture we have obtained. We are also interested in finding how many levels of unshared cache are necessary as the number of processor increases to see whether or not we can justify the cost of adding more memory and levels of memory to maintain performance. We also like to study the cache performance using multithreaded benchmarks by varying the cache size and set associativity as multithreaded benchmarks in general require more memory bandwidth compared to single threaded benchmarks, hence they should be more sensitive to an increase in cache size and set associativity. We will also look into using a split transaction bus technique as we believe that we can improve the overall performance compared to the results from Chapter 4 using a single bus. In a split transaction bus, a transaction is split into two transactions: request and reply.

A processor can request something and releases the bus when it is stalled so that others can use it and receive the response later, hence more memory bandwidth for the system and hopefully better bus utilization. The design will be more complex than non-split bus architecture but we hope that we would get a significant performance boost to justify the complexity of the design.

CHAPTER VI

CONCLUSION

This dissertation has provided a study on the effect of increasing the number of processors to a shared bus. In sharing a bus, two factors determine the overall processor and cache performance and they are bus contention and memory thrashing. Based on our research, we have concluded that by keeping a constant ratio between the numbers of processors to the shared cache size, we have prevented memory thrashing from causing significant performance loss. The bus contention however cannot be prevented and interferes with the overall performance. Sharing a L2 cache is less desirable compared to sharing a L3 cache or using hierarchical architecture because the performance drops at a higher rate as the number of processors increases. Abakus uses a scalar processor while commercial processors use superscalar with higher memory bandwidth required per processor, hence we expect that the shared bus contention may hit 100% well before sharing a L2 cache with sixteen superscalar processors. A more exotic architecture needs to be researched and developed to handle the contention on a shared bus.

REFERENCES

[1] Z. Chuanjun, "Balanced instruction cache: reducing conflict misses of direct-mapped caches through balanced subarray accesses," Computer Architecture Letters, vol. 5, pp. 2-5, 2006.

[2] A. Agarwal and S. D. Pudar, "Column-associative Caches: A Technique For Reducing The Miss Rate Of Direct-mapped Caches," in Computer Architecture, 1993, Proceedings of the 20th Annual International Symposium on, 1993, pp. 179-190.

[3] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on, 1990, pp. 364-373.

[4] Y.C. Maa, D. K. Pradhan, and D. Thiebaut, "Two economical directory schemes for large-scale cache coherent multiprocessors," SIGARCH Comput. Archit. News, vol. 19, p. 10, 1991.

[5] T. Sasaki, T. Inoue, N. Omori, T. Hironaka, H. J. Mattausch, and T. Koide, "Chip size and performance evaluations of shared cache for on-chip multiprocessor," Syst. Comput. Japan, vol. 36, pp. 1-13, 2005.

[6] Z. Wang, Q. Zuo, and J. Li, "An Intelligent Multi-Port Memory," in Intelligent Information Technology Application Workshops, 2008. IITAW '08. International Symposium on, 2008, pp. 251-254.

[7] L. Caixia, L. Jiaxin, Z. Hongli, and Z. Qi, "HHMA: A Hierarchical Hybrid Memory Architecture Sharing Multi-Port Memory," in Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for, 2008, pp. 1320-1325.

[8] J. Weixing, S. Feng, Q. Baojun, and S. Hong, "Multi-port Memory Design Methodology Based on Block Read and Write," in Control and Automation, 2007. ICCA 2007. IEEE International Conference on, 2007, pp. 256-259.

[9] S. Shiratake, K. Tsuchida, H. Toda, H. Kuyama, M. Wada, F. Kouno, T. Inaba, H. Akita, and K. Isobe, "A pseudo multi-bank DRAM with categorized access sequence," in VLSI Circuits, 1999. Digest of Technical Papers. 1999 Symposium on, 1999, pp. 127-130.

[10] Y. Mukuda, K. Aoyama, K. Johguchi, H. J. Mattausch, and T. Koide, "Access Queues for Multi-Bank Register Files Enabling Enhanced Performance of Highly Parallel Processors," in TENCON 2006. 2006 IEEE Region 10 Conference, 2006, pp. 1-4.

[11] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware Dynamic Cache Partitioning for Multicore Architectures," in Parallel Processing, 2009. ICPP '09. International Conference on, 2009, pp. 18-25.

[12] J. H. Tseng and K. Asanovic, "Banked multiported register files for high-frequency superscalar microprocessors," in Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on, 2003, pp. 62-71.

[13] T. Hironaka, M. Maeda, K. Tanigawa, T. Sueyoshi, K. Aoyama, T. Koide, H. J. Mattausch, and T. Saito, "Superscalar processor with multi-bank register file," in Innovative Architecture for Future Generation High-Performance Processors and Systems, 2005, 2005, p. 10 pp.

[14] S. Cho, "I-cache multi-banking and vertical interleaving," presented at the Proceedings of the 17th ACM Great Lakes symposium on VLSI, Stresa-Lago Maggiore, Italy, 2007.

[15] S. Bieschewski, J. M. Parcerisa, and A. Gonzalez, "Memory bank predictors," in Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on, 2005, pp. 666-668.

[16] J. Koh, A. Ken-ichi, S. Tetsuya, M. Hans Jurgen, K. Tetsushi, M. Moto, H. Tetsuo, and T. Kazuya, "Multi-Bank Register File for Increased Performance of Highly-Parallel Processors," in Solid-State Circuits Conference, 2006. ESSCIRC 2006. Proceedings of the 32nd European, 2006, pp. 154-157.

[17] T. Saito, M. Maeda, T. Hironaka, K. Tanigawa, T. Sueyoshi, K. Aoyama, T. Koide, and H. J. Mattausch, "Design of superscalar processor with multi-bank register file," in Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on, 2005, vol. 4, pp. 3507-3510.

[18] T. Yamauchi, L. Hammond, and K. Olukotun, "The hierarchical multi-bank DRAM: a high-performance architecture for memory integrated with processors," in Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on, 1997, pp. 303-319.

[19] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling," in Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on, 2005, pp. 408-419.

[20] J. C. Villanueva, J. Flich, J. Duato, H. Eberle, N. Gura, and W. Olesinski, "A performance evaluation of 2D-mesh, ring, and crossbar interconnects for chip multi-processors," in Network on Chip Architectures, 2009. NoCArc 2009. 2nd International Workshop on, 2009, pp. 51-56.

[21] S. Murali, L. Benini, and G. De Micheli, "An Application-Specific Design Methodology for On-Chip Crossbar Generation," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, pp. 1283-1296, 2007.

[22] K. Johguchi, Z. Zhu, T. Hirakawa, T. Koide, T. Hironaka, and H. J. Mattausch, "Distributed crossbar architecture for area-efficient combined data/instruction caches with multiple ports," Electronics Letters, vol. 40, pp. 160-162, 2004.

[23] Y. Aridor, T. Domany, O. Goldshmidt, Y. Kliteynik, E. Shmueli, and J. E. Moreira, "Multitoroidal Interconnects For Tightly Coupled Supercomputers," Parallel and Distributed Systems, IEEE Transactions on, vol. 19, pp. 52-65, 2008.

[24] C. Shu-Hsuan, C. Chien-Chih, W. Chi-Neng, C. Yi-Chao, C. Tien-Fu, W. Chao-Ching, and W. Jinn-Shyan, "No cache-coherence: A single-cycle ring interconnection for multi-core L1-NUCA sharing on 3D chips," in Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE, 2009, pp. 587-592.

[25] H. Gang, R. H. Klenke, and J. H. Aylor, "Performance modeling of hierarchical crossbar-based multicomputer systems," Computers, IEEE Transactions on, vol. 50, pp. 877-890, 2001.

[26] Z. Ying Ping, J. Taikyeong, C. Fei, W. Haiping, R. Nitzsche, and G. R. Gao, "A study of the on-chip interconnection network for the IBM Cyclops64 multi-core architecture," in Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 2006, pp. 10.

[27] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz Mesh Interconnect for a Teraflops Processor," Micro, IEEE, vol. 27, pp. 51-61, 2007.

[28] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, B. Liewei, J. Brown, M. Mattina, M. Chyi-Chang, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," in Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International, 2008, pp. 88-598.

[29] P. Cheolmin, R. Badeau, L. Biro, J. Chang, T. Singh, J. Vash, W. Bo, and T. Wang, "A 1.2 TB/s on-chip ring interconnect for 45nm 8-core enterprise Xeon®processor," in Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International, 2010, pp. 180-181.

[30] S. Murali, L. Benini, and G. De Micheli, "An Application-Specific Design Methodology for On-Chip Crossbar Generation," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, pp. 1283-1296, 2007.

[31] K. Johguchi, Z. Zhu, T. Hirakawa, T. Koide, T. Hironaka, and H. J. Mattausch, "Distributed crossbar architecture for area-efficient combined data/instruction caches with multiple ports," Electronics Letters, vol. 40, pp. 160-162, 2004.

[32] Y. Aridor, T. Domany, O. Goldshmidt, Y. Kliteynik, E. Shmueli, and J. E. Moreira, "Multitoroidal Interconnects For Tightly Coupled Supercomputers," Parallel and Distributed Systems, IEEE Transactions on, vol. 19, pp. 52-65, 2008.

[33] Deborah A.Wallach, "PHD: A Hierarchical Cache Coherent Protocol", master thesis, MIT, September 1992 .

[34] C. Anderson and J. L. Baer, "A multi-level hierarchical cache coherence protocol for multiprocessors," in Parallel Processing Symposium, 1993., Proceedings of Seventh International, 1993, pp. 142-148.

[35] B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," Computer, vol. 24, pp. 52-60, 1991.

[36] Q. Yang, "Performance analysis of a cache-coherent multiprocessor based on hierarchical multiple buses," in Databases, Parallel Architectures and Their Applications, PARBASE-90, International Conference on, 1990, pp. 248-257.

[37] G. Dewan and P. Biswas, "A snooping cache coherency protocol for hierarchically organized multiprocessors," Microprocess. Microprogram., vol. 31, pp. 105-111, 1991.

[38] J.S. Vitter and E.A.M. Shriver, "Algorithms for Parallel Memory II: Hierarchical Multilevel Memories," Algorithmica, vol. 12, pp. 148-169, 1994.

[39] Lioupis, D., and Milios, S. Exploring cache performance in multithreaded processors, Microprocessors and Microsystems, vol.20, no.10, Jun. 1997.

[40] J.-L. Baer and W.-H. Wang, "Multilevel cache hierarchies: organizations, protocols, and performance," J. Parallel Distrib. Comput., vol. 6, pp. 451-476, 1989.

[41] M. K. Vernon, R. Jog, and G. S. Sohi, "Performance analysis of hierarchical cache-consistent multiprocessors," Perform. Eval., vol. 9, pp. 287-302, 1989.

[42] J. Bertoni, J.-L. Baer, and W.-H. Wang, "Scaling shared-bus multi-processors with multiple buses and shared caches: a performance study," Microprocess. Microsyst., vol. 16, pp. 339-350, 1992.

[43] C. Anderson and J. L. Baer, "Two techniques for improving performance on bus-based multiprocessors," in High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on, 1995, pp. 264-275.

[44] F. N. Sibai, "On the performance benefits of sharing and privatizing second and third-level cache memories in homogeneous multi-core architectures," Microprocess. Microsyst., vol. 32, pp. 405-412, 2008.

[45] P. Jin Young and L. Choi, "RING-DATA ORDER: A new cache coherence protocol for ring-based multicores," in High Performance Computing & Simulation, 2009. HPCS '09. International Conference on, 2009, pp. 82-88.

[46] A. Jaleel, "Memory Characterization of Workloads Using Instrumentation-Driven Simulation," http://www.glue.umd.edu '~ajaleel/workload, 2007.

[47] A. Jaleel, R. S. Cohn, C. K. Luk, B. L. Jacob. "CMP$im: Using PIN to Characterize Memory Behavior of Emerging Workloads on CMPs", Technical Report - UMD-SCA-2006-01.

[48] P. Lu, P. Jih-Kwon, T. K. Prakash, C. Yen-Kuang, and D. Koppelman, "Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study," in Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa, 2007, pp. 55-64.

[49] T. K. Prakash, "Performance Analysis of Intel Core 2 Duo Processor," Master's thesis, Louisiana State University, 2007.

[50] K. Ganesan, D. Panwar, and L. K. John, "Generation, Validation and Analysis of SPEC CPU2006 Simulation Points Based on Branch, Memory and TLB Characteristics," presented at the Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking, Austin, TX, 2009.

[51] L. Shengmei, Q. Lin, T. Zhizhong, C. Buqi, and G. Xingyu, "Performance Characterization of SPEC CPU2006 Benchmarks on Intel and AMD Platform," in Education Technology and Computer Science, 2009. ETCS '09. First International Workshop on, 2009, pp. 116-121.

[52] A. A. Nair and L. K. John, "Simulation points for SPEC CPU 2006," in Computer Design, 2008. ICCD 2008. IEEE International Conference on, 2008, pp. 397-403.

[53] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru, "Performance characterization of SPEC CPU benchmarks on Intel's core microarchitecture based processor," in SPEC Benchmark Workshop, 2007.

APPPENDICES

APPENDIX A. INSTALLING MIPS CROSS COMPILER AND CROSS COMPILING SPEC CPU 2006 BENCHMARKS TO MIPS

A.1 Installing MIPS Cross Compiler

1. You need to be in bash ($) and 'bison' and 'flex' are pre-requisites
2. Do $ mkdir mycrosstoolbuild
3. Do $ cd mycrosstoolbuild
4. Do $ wget http://kegel.com/crosstool/crosstool-0.43.tar.gz
5. Do $ tar xzvf crosstool-0.43.tar.gz
6. Do $ cd crosstool-0.43
7. Edit demo-mipsel.sh to reflect the following (Adjust PARALLELMFLAGS to reflect the number of CPUs on the build system):

```
set -ex
TARBALLS_DIR=$HOME/downloads
RESULT_TOP=$HOME/crosstool
export TARBALLS_DIR RESULT_TOP
GCC_LANGUAGES="c,c++"
export GCC_LANGUAGES
PARALLELMFLAGS="-j4"
export PARALLELMFLAGS
```

8. Do $ vi gcc-3.4.5-glibc-2.3.6-tls.dat
9. Update GLIBC_EXTRA_CONFIG to GLIBC_EXTRA_CONFIG="$GLIBC_EXTRA_CONFIG --with-tls --with-__thread --without-fp --enable-kernel=2.4.18"
10. Do $ vi mipsel.dat (Do $ vi mips.dat if you want to have big endian)

11. Update GCC_EXTRA_CONFIG to GCC_EXTRA_CONFIG="$GCC_EXTRA_CONFIG --with-float=soft"
12. Download glibc-2.3.6-csuMakefile-patch from https://eng.ucmerced.edu/root01/SoE_April_07/ComputingSupport/il/collaboratory/collab-software/compilers-and-interpreters/glibc-2.3.6-csuMakefile.patch
13. Place the patch into: ~/yourusername/mycrosstoolbuild/crosstool/patches/glibc-2.3.6
14. Go back to ~/yourusername/mycrosstoolbuild/crosstool
15. Do $ bash
16. Do $ unset LD_LIBRARY_PATH (everytime you want to **build** something new, ALWAYS do unset)
17. Do $ sh demo-mips.sh (to run the script as this will take a while)
18. Now go to your /home/yourusername/crosstool (Do $ cd followed by $ cd crosstool)
19. Do $ mkdir ccmipsel
20. Do $ mv gcc-3.4.5-glibc-2.3.6/mips-unknown-linux-gnu ccmipsel
21. Do $ rm –r gcc-3.4.5-glibc-2.3.6
22. Do $ export PATH=${PATH}:/home/yourusername/crosstool/ccmipsel/mipsel-unknown-linux-gnu/bin
    (You MUST do this on every **new** console/session you use/have)
23. Example: $ mipsel-unknown-linux-gnu-gcc -O2 -static -msoft-float -o mybinary mybinary.c -lm
24. Example: $ file mybinary
25. Example: $ mipsel-unknown-linux-gnu-objdump -D mybinary > mybinary.txt

Sources:

http://www.kegel.com/crosstool/

https://eng.ucmerced.edu/root01/SoE_April_07/ComputingSupport/il/collaboratory/collab-software/compilers-and-interpreters/mips-cross-compiler-package

A.2 Cross Compile Spec 2006 Benchmarks to MIPS:

1. Install Spec 2006 in /home/yourusername/SPEC2006/
2. If you are using Unix/Linux go to : /home/yourusername/SPEC2006/ and type in bash

   $ **. ./shrc** (yes, it is dot then space then dot then forward slash then shrc)

3. Check the following website for the list of Spec 2006 benchmark:

   http://www.spec.org/cpu2006/Docs/

4. Do $ cd /home/yourusername/SPEC2006/config
5. Do $ vi linux32-i386-gcc42.cfg (if you have vim, use it)
6. Edit it so it looks like the following (assuming you want to disable floating point):

   CC = /home/yourusername/crosstool/ccmipsel/mipsel-unknown-linux-gnu/bin/mipsel-unknown-linux-gnu-gcc

   CXX = /home/yourusername/crosstool/ccmipsel/mipsel-unknown-linux-gnu/bin/mipsel-unknown-linux-gnu-g++

   FC = /usr/local/gcc42-0715-32/bin/gfortran

   COPTIMIZE = -O2 -static -msoft-float

   CXXOPTIMIZE = -O2 -static -msoft-float

   FOPTIMIZE = -O2

   (take out -msoft-float if you want to have floating point number)

7. Do $ cd /home/yourusername/SPEC2006/bin
8. Do $ runspec --config= linux32-i386-gcc42.cfg --action=build --tune=base bzip2

   Or you can also type in the benchmark number, in this case replace **bzip2** with **401**

9. Get your binary in /home/yourusername/SPEC2006/benchspec/CPU2006/401.bzip2/run/

   If this is your first time doing it, it should be in:

   /home/yourusername/SPEC2006/benchspec/CPU2006/401.bzip2/run/build_base_i386-m32-gcc42-nn.0000

Source: http://gem5.org/SPEC2006_benchmarks

Check out the following links should you encounter an error:

http://www.spec.org/cpu2006/Docs/runspec.html#section2.4

http://www.spec.org/cpu2006/Docs/faq.html

APPENDIX B. EXAMPLES IN USING ABAKUS TO CREATE A QUAD-CORE ARCHITECTURE USING A PRIVATE 32KB L1 CACHE FOR EACH PROCESSOR AND A SHARED L2 2MB CACHE

Download Abakus and untar it to a folder using tar –zxvf command. Get in to the folder and type 'make'; this will build Abakus. Once done, go to the ~/src/examples/dual_mips_shared_l2 folder and change the mips_l1.cpp, dual_mips_shared_l2.cpp, dual_mips_shared_l2.h, and testbench.cpp file as shown in Appendix B1 and B2. Go back to the main Abakus directory and type 'make'. You will find your executable in ~/src/examples/dual_mips_shared_l2.

B.1 Instructions To Create a Quad-Core Using a Shared L2 Bus

Open your mips_l1.cpp and find the following 2 – 3 lines:

```
icache("icache", 1, mem_manager, 1L << 15, 8L, 8),
dcache("dcache", 1, mem_manager, 1L << 15, 8L, 8,
        2.0*1.0, 2.0*1.0, ab_time_unit),
```

"1L << 15" means that you are creating an L1 cache with a size of 2^ 15 = 32KB.

"8L" means that you are going to have 8 lines for your cache, and the last "8" means 8 way.

Make your changes to mips_l1.cpp, save it and open dual_mips_shared_l2.cpp and find any lines that deals with p0 or p1 and expands them to p2 or p3, for example:

The lines:

```
p0_req12i("p0_req12i"),
p0_grant12i("p0_grant12i"),
p0_req12d("p0_req12d"),
p0_grant12d("p0_grant12d"),
p1_req12i("p1_req12i"),
p1_grant12i("p1_grant12i"),
p1_req12d("p1_req12d"),
p1_grant12d("p1_grant12d"),
```

need to be expanded to:

p0_req12i("p0_req12i"),

p0_grant12i("p0_grant12i"),

p0_req12d("p0_req12d"),

p0_grant12d("p0_grant12d"),

p1_req12i("p1_req12i"),

p1_grant12i("p1_grant12i"),

p1_req12d("p1_req12d"),

p1_grant12d("p1_grant12d"),

p2_req12i("p2_req12i"),

p2_grant12i("p2_grant12i"),

p2_req12d("p2_req12d"),

p2_grant12d("p2_grant12d"),

p3_req12i("p3_req12i"),

p3_grant12i("p3_grant12i"),

p3_req12d("p3_req12d"),

p3_grant12d("p3_grant12d"),

Make your changes to dual_mips_shared_l2.cpp, save it and open dual_mips_shared_l2.h and find any lines that deals with p0 or p1 and expands them to p2 or p3, for example:

The lines:

ab_signal<bool> p0_req12i;

ab_signal<bool> p0_grant12i;

ab_signal<bool> p0_req12d;

ab_signal<bool> p0_grant12d;

ab_signal<bool> p1_req12i;

ab_signal<bool> p1_grant12i;

ab_signal<bool> p1_req12d;

ab_signal<bool> p1_grant12d;

need to be expanded to:

```
ab_signal<bool> p0_req12i;
ab_signal<bool> p0_grant12i;
ab_signal<bool> p0_req12d;
ab_signal<bool> p0_grant12d;
ab_signal<bool> p1_req12i;
ab_signal<bool> p1_grant12i;
ab_signal<bool> p1_req12d;
ab_signal<bool> p1_grant12d;
ab_signal<bool> p2_req12i;
ab_signal<bool> p2_grant12i;
ab_signal<bool> p2_req12d;
ab_signal<bool> p2_grant12d;
ab_signal<bool> p3_req12i;
ab_signal<bool> p3_grant12i;
ab_signal<bool> p3_req12d;
ab_signal<bool> p3_grant12d;
```

Make your changes to dual_mips_shared_l2.h, save it and open testbench.cpp and find any lines that deals with p0 or p1 and expands them to p2 or p3, for example:

The lines:

```
top.p0.dump(dumpfile);
top.p1.dump(dumpfile);
```

need to be expanded to:

```
top.p0.dump(dumpfile);
top.p1.dump(dumpfile);
top.p2.dump(dumpfile);
top.p3.dump(dumpfile);
```

Appendix B.2 contains all the changes needed to create a quad-core architecture.

B.2Quad-Core Using a Shared L2 Bus

Replace your mips_l1.cpp with the following

```
#include "mips_l1.h"
#include <fcntl.h>
#include <unistd.h>

mips_l1::mips_l1(sc_module_name name,
    ab_host_mem_manager *mem_manager_,
    int pid,
    addr_t ptag_inst,
    addr_t ptag_data
):
    sc_module(name),
    mem_manager(mem_manager_),
    //local channel initialization
    inst_stall("inst_stall"),
    data_stall("data_stall"),
    dreq("dreq"),
    dgrant("dgrant"),
    dbus("dbus", 4),
    //sub-module instance initialization
    p("p", pid, ptag_inst, ptag_data),
    inst_stall_length("inst_stall_length"),
    data_stall_length("data_stall_length"),
    icache("icache", 1, mem_manager, 1L << 15, 8L, 8),
```

```
    icache_miss_count("icache_miss_count"),
    dcache_cpu("dcache_cpu"),
    dcache_access_count("dcache_access_count"),
    dcache_miss_count("dcache_miss_count"),
    dcache("dcache", 1, mem_manager, 1L << 15, 8L, 8,
            2.0*1.0, 2.0*1.0, ab_time_unit),
    interactive(0)
{
    //sub-module connections
    p.clk(clk);
    p.inst_mem(icache);
    p.inst_stall(inst_stall);
    p.data_mem(dcache_cpu);
    p.data_stall(data_stall);
    inst_stall_length.start(inst_stall);
    inst_stall_length.stop(inst_stall);
    data_stall_length.start(data_stall);
    data_stall_length.stop(data_stall);
    icache.stall(inst_stall);
    dcache_cpu.stall(data_stall);
    dcache_cpu.client_port_con(dbus.con);
    dcache_cpu.client_port(dbus);
    dcache_cpu.req(dreq);
    dcache_cpu.grant(dgrant);
    icache_miss_count.clk(icache.miss);
    icache_miss_count.cond(icache.miss);
```

```
    dcache_access_count.clk(dreq);
    dcache_access_count.cond(dreq);
    dcache_miss_count.clk(dcache.miss);
    dcache_miss_count.cond(dcache.miss);
    dcache.master_port_con(dbus.con);
    dcache.master_port(dbus);
    dcache.master_req(dreq);
    dcache.master_grant(dgrant);
    icache.client_port_con(mem_bus_con);
    dcache.client_port_con(mem_bus_con);
    icache.client_port(mem_bus);
    dcache.client_port(mem_bus);
    icache.client_req(req12i);
    dcache.client_req(req12d);
    icache.client_grant(grant12i);
    dcache.client_grant(grant12d);
}

void mips_l1::initialize(ab_main_mem *main_mem,
    int argc, char* *argv, char* *envp)
{
    addr_t pc_init_value = 0; //just in case its not initialized by loader
    addr_t brk_point = 0;
    if (argc > 0) {
        //argv should point to program name to load
        main_mem->mem.load(argv[0], p.ptag_inst, pc_init_value, brk_point);
```

```
}
p.initialize(pc_init_value, brk_point);


//redirect io
bool change = 1;
while (change) {
   change = 0;
   int i = argc - 2;
       if (argv[i][0] == '<') { //redirect stdin
          int fd = open(argv[i+1], O_RDONLY);
          if (fd < 0) {
             cout << "could not open file " << argv[i+1] << endl;
          }
          p.except_handler.ioredirect[0] = fd;
          argc = i;
      change = 1;
          cout << name() << " redirecting input to " << argv[i+1] << endl;
       }
   if (argv[i][0] == '>') { //redirect stdout
      int fd = open(argv[i+1], O_WRONLY | O_CREAT, 00644);
      if (fd < 0) {
         cout << "could not open file " << argv[i+1] << endl;
      }
      p.except_handler.ioredirect[1] = fd;
      argc = i;
      change = 1;
```

```
            cout << name() << " redirecting output to " << argv[i+1] << endl;
    }
}

//initialize stack
addr_t stack_ptr = mips32int::STACK_BASE - mips32int::MAX_ENVIRON;
p.gpr.chan.write(29, stack_ptr); //sp (stack pointer)

/*write argc to stack*/
paddr_t ptag = ((paddr_t) p.ptag_inst) << 56;
main_mem->mem.write_mem(0, ptag + ((paddr_t) stack_ptr),
    sizeof(addr_t), (cblock_t) &argc, sizeof(addr_t) );
p.gpr.chan.write(4, argc);
stack_ptr += sizeof(addr_t);


/*skip stack_ptr past argv pointer array*/
addr_t argAddr = stack_ptr;
p.gpr.chan.write(5, argAddr);
stack_ptr += (argc+1)*sizeof(addr_t);

/*skip env pointer array*/
addr_t envAddr = stack_ptr;
for (int i=0; envp[i]; i++)
        stack_ptr += 4;
stack_ptr += 4;
```

```
/*write argv to stack*/
for (int i=0; i<argc; i++) {
   main_mem->mem.write_mem(0, ptag + ((paddr_t) (argAddr+i*sizeof(addr_t))),
      sizeof(addr_t), (cblock_t) &stack_ptr, sizeof(addr_t));
   for (int j = 0; argv[i][j] != '\0'; j++)
      main_mem->mem.write_mem(0, ptag + ((paddr_t) (stack_ptr+j)),
         1, (cblock_t) &argv[i][j], 1);
   /*0 already at the end of the string as done by initialization*/
   stack_ptr += strlen(argv[i])+1;
}

/*0 already at the end argv pointer array*/

/*write env to stack*/
for (int i=0; envp[i]; i++) {
   main_mem->mem.write_mem(0, ptag + ((paddr_t) (envAddr+i*sizeof(addr_t))),
      sizeof(addr_t), (cblock_t) &stack_ptr, sizeof(addr_t) );
   for (int j = 0; envp[i][j] != '\0'; j++)
      main_mem->mem.write_mem(0, ptag + ((paddr_t) (stack_ptr+j)),
         1, (cblock_t) &envp[i][j], 1);
   /*0 already at the end of the string as done by initialization*/
   stack_ptr += strlen(envp[i])+1;
}

/*stack overflow*/
```

```
    if (stack_ptr+sizeof(addr_t)>=p.STACK_BASE) {
        cout << "Environment overflow for processor " << p.pid
            << ". Need to increase MAX_ENVIRON.\n";
        SC_REPORT_ERROR("abakus", name());
    }
}

void mips_l1::evaluate() {
    p.evaluate();
}

void mips_l1::evaluate_end() {
    p.evaluate_end();
    inst_stall_length.evaluate_start();
    inst_stall_length.evaluate_stop();

    //performance measures
    if (p.controller.stall_if_chan.read() == 1)
        if_stall_count += 1;
    else {
        if ( (p.controller.pc_inst_reg.chan.read() )->icode == 0 )
            if_nop_count += 1;
        else
            if_icount++;
    }
    if (p.controller.stall_wr_back_chan.read() == 1)
```

```
        wb_stall_count += 1;
    else {
        if ( (p.controller.mem_wb_inst_reg.chan.read() )->icode == 0 )
            wb_nop_count += 1;
        else
            icount++;
    }
    if (inst_stall.read()) inst_stall_count++;
    if (data_stall.read()) data_stall_count++;
    if (inst_stall.read() && data_stall.read()) inst_data_stall_count++;

    if (interactive) {
        instruction *inst = p.controller.mem_wb_inst_reg.chan.read();
        addr_t addr = inst->iaddr;
        icode_t code = inst->icode;
        debug_monitor(addr, code);
    }
}

void mips_l1::perf_sum() {
    cout << endl;
    cout << "processor " << p.pid << " performance summary" << endl;
    cout << "write back NOP count: " << wb_nop_count << endl;
    cout << "write back stall count: " << wb_stall_count << endl;
    cout << "instruction count: " << icount << endl;
    cout << "IPC: " << ((float) icount)/((float) ab_clk_count) << endl;
```

```
cout << "fetch NOP count: " << if_nop_count << endl;
cout << "fetch stall count: " << if_stall_count << endl;
cout << "fetch instruction count: " << if_icount << endl;
cout << endl;
cout << "i-cache accesses: " << if_nop_count + if_icount
   << " misses: " << icache_miss_count.count
   << " miss rate " << (float) icache_miss_count.count
         / (float) (if_nop_count + if_icount) << endl;
cout << "d-cache accesses: " << dcache_access_count.count
   << " misses: " << dcache_miss_count.count
   << " miss rate " << (float) dcache_miss_count.count
         / (float) dcache_access_count.count << endl;

float Pstall_icache = (float) inst_stall_count / (float) ab_clk_count;
cout << "i-cache stall cycles: " << inst_stall_count
    << " probability of stalled i-cache: " << Pstall_icache << endl;

float Pstall_dcache = (float) data_stall_count / (float) ab_clk_count;
cout << "d-cache stall cycles: " << data_stall_count
    << " probability of stalled d-cache: " << Pstall_dcache << endl;

float Pstall_idcache = (float) inst_data_stall_count/ (float) ab_clk_count;
float COV_stall_idcache = Pstall_idcache - Pstall_icache * Pstall_dcache;
float CORR_stall_idcache = COV_stall_idcache/sqrt( Pstall_icache
   * (1. - Pstall_icache) * Pstall_dcache * (1. - Pstall_dcache) );
cout << "i and d cache stall cycles: " << inst_data_stall_count
```

```
        << " probability of stalled i and dcache: " << Pstall_idcache
        << " covariance: " << COV_stall_idcache
        << " correlation: " << CORR_stall_idcache << endl;
    cout << endl;

    inst_stall_length.dump(cout);
    data_stall_length.dump(cout);
    cout << endl;

}

void mips_l1::dump(ostream &out) const {
    out << endl << name() << endl;
    out << "instruction count: " << icount << endl;

    p.dump(out);
    icache.dump(out);
    req12i.dump(out);
    grant12i.dump(out);
    dcache.dump(out);
    req12d.dump(out);
    grant12d.dump(out);
}
```

Replace your dual_mips_shared_l2.cpp with the following:

```
#include "dual_mips_shared_l2.h"

int npid = 0; //incremented be each processor instance

void parse(char *command, int &pargc, char **pargv) {
    const int MAXARGS = 10;
    const int MAXCHARS = 80;
    pargc = 0;
    int i = 0;
    bool word_started = 0;
    cin.get(command[0]);
    while (command[i] != '\n') {
        if (command[i] == ' ') {
            while (cin.peek() == ' ') cin.get(command[i]);
            if (word_started) {
                command[i] = '\0';
                    word_started = 0;
            }
            //blanks ignored if not word_started
        } else { //non-blank
            if (!word_started) {
                pargv[pargc++] = &command[i];
                if (pargc >= MAXARGS) {
                    cout << "too many arguments" << endl;
```

```
                SC_REPORT_ERROR("abakus", "parse error");
            }
                word_started = 1;
            }
            //no new arg if word_started already
        }
        if (++i >= MAXCHARS) {
            cout << "too many characters" << endl;
            SC_REPORT_ERROR("abakus", "parse error");
        }
        cin.get(command[i]);
    }
    command[i] = '\0';
    if (pargc == 0) {
        cout << "usage: [mips-elf-executable-file]"
<< " [arguments to executable file]" << endl;
      SC_REPORT_ERROR("abakus", "parse error");
    }
}

dual_mips_shared_l2::dual_mips_shared_l2(sc_module_name name)
    : sc_module(name),
    //local channel initialization
    p0_req12i("p0_req12i"),
    p0_grant12i("p0_grant12i"),
    p0_req12d("p0_req12d"),
```

```
p0_grant12d("p0_grant12d"),
p1_req12i("p1_req12i"),
p1_grant12i("p1_grant12i"),
p1_req12d("p1_req12d"),
p1_grant12d("p1_grant12d"),

p2_req12i("p2_req12i"),
p2_grant12i("p2_grant12i"),
p2_req12d("p2_req12d"),
p2_grant12d("p2_grant12d"),
p3_req12i("p3_req12i"),
p3_grant12i("p3_grant12i"),
p3_req12d("p3_req12d"),
p3_grant12d("p3_grant12d"),

req12("req12"),
grant12("grant12"),
req23("req23"),
grant23("grant23"),
bus1("bus1", 64),
bus2("bus2", 64),
//sub-module instance initialization
mem_manager("mem_manager", 1 << 28),

p0("p0", &mem_manager, 0, 1, 1),
p1("p1", &mem_manager, 1, 2, 2),
```

```
    p2("p2", &mem_manager, 2, 3, 3),
    p3("p3", &mem_manager, 3, 4, 4),

    arbiter("arbiter", round_robin),
    cache2("cache2", 1, &mem_manager, 1L << 21, 8L, 8,
           2.0*3.0, 2.0*2.0, ab_time_unit),
    cache2_access_count("cache2_access_count"),
    cache2_miss_count("cache2_miss_count"),
    main_mem("main_mem", 0, &mem_manager, 8, 2.0*10.0, 2.0*8.0, ab_time_unit),
    main_mem_access_count("main_mem_access_count"),
    bus1_busy_count(0)
{
    //sub-module connections
    p0.clk(clk);
    p0.mem_bus_con(bus1.con);
    p0.mem_bus(bus1);
    p0.req12i(p0_req12i);
    p0.req12d(p0_req12d);
    p0.grant12i(p0_grant12i);
    p0.grant12d(p0_grant12d);

    p1.clk(clk);
    p1.mem_bus_con(bus1.con);
    p1.mem_bus(bus1);
    p1.req12i(p1_req12i);
```

```
p1.req12d(p1_req12d);
p1.grant12i(p1_grant12i);
p1.grant12d(p1_grant12d);

p2.clk(clk);
p2.mem_bus_con(bus1.con);
p2.mem_bus(bus1);
p2.req12i(p2_req12i);
p2.req12d(p2_req12d);
p2.grant12i(p2_grant12i);
p2.grant12d(p2_grant12d);

p3.clk(clk);
p3.mem_bus_con(bus1.con);
p3.mem_bus(bus1);
p3.req12i(p3_req12i);
p3.req12d(p3_req12d);
p3.grant12i(p3_grant12i);
p3.grant12d(p3_grant12d);

arbiter.req_client(p0_req12i);
arbiter.grant_client(p0_grant12i);
arbiter.req_client(p1_req12i);
arbiter.grant_client(p1_grant12i);
arbiter.req_client(p2_req12i);
arbiter.grant_client(p2_grant12i);
```

```
arbiter.req_client(p3_req12i);
arbiter.grant_client(p3_grant12i);

arbiter.req_client(p0_req12d);
arbiter.grant_client(p0_grant12d);
arbiter.req_client(p1_req12d);
arbiter.grant_client(p1_grant12d);
arbiter.req_client(p2_req12d);
arbiter.grant_client(p2_grant12d);
arbiter.req_client(p3_req12d);
arbiter.grant_client(p3_grant12d);
arbiter.req_master(req12);
arbiter.grant_master(grant12);

cache2.master_port_con(bus1.con);
cache2.master_port(bus1);
cache2.master_req(req12);
cache2.master_grant(grant12);
cache2.client_port_con(bus2.con);
cache2.client_port(bus2);
cache2.client_req(req23);
cache2.client_grant(grant23);
cache2_access_count.clk(req12);
cache2_access_count.cond(req12);
cache2_miss_count.clk(cache2.miss);
cache2_miss_count.cond(cache2.miss);
```

```
    main_mem.master_port_con(bus2.con);
    main_mem.master_port(bus2);
    main_mem.master_req(req23);
    main_mem.master_grant(grant23);
    main_mem_access_count.clk(req23);
    main_mem_access_count.cond(req23);
}

void dual_mips_shared_l2::initialize(char **envp) {
    char command[80];
    int pargc;
    char* pargv[10];

    //initialize processor pointer array
    pl1ptr = new mips_l1*[npid];
    pl1ptr[p0.p.pid] = &p0;
    pl1ptr[p1.p.pid] = &p1;
    pl1ptr[p2.p.pid] = &p2;
    pl1ptr[p3.p.pid] = &p3;

    //sanity check
    cout << "(pl1ptr[0]->p).pid = " << (pl1ptr[0]->p).pid << endl;
    cout << "(pl1ptr[1]->p).pid = " << (pl1ptr[1]->p).pid << endl;
    cout << "(pl1ptr[2]->p).pid = " << (pl1ptr[2]->p).pid << endl;
    cout << "(pl1ptr[3]->p).pid = " << (pl1ptr[3]->p).pid << endl;
```

```
cout << endl;

cout << "p" << p0.p.pid << " command line: ";
parse(command, pargc, pargv);
for (int i = 0; i < pargc; i++) {
        cout << "argv[" << i << "] = ";
        for (int j = 0; pargv[i][j] != '\0'; j++) {
            cout << pargv[i][j];
        }
   cout << endl;
}
p0.initialize(&main_mem, pargc, pargv, envp);

cout << "p" << p1.p.pid << " command line: ";
parse(command, pargc, pargv);
for (int i = 0; i < pargc; i++) {
        cout << "argv[" << i << "] = ";
        for (int j = 0; pargv[i][j] != '\0'; j++) {
            cout << pargv[i][j];
        }
   cout << endl;
}
p1.initialize(&main_mem, pargc, pargv, envp);

cout << "p" << p2.p.pid << " command line: ";
parse(command, pargc, pargv);
```

```
    for (int i = 0; i < pargc; i++) {
        cout << "argv[" << i << "] = ";
        for (int j = 0; pargv[i][j] != '\0'; j++) {
            cout << pargv[i][j];
        }
        cout << endl;
    }
    p2.initialize(&main_mem, pargc, pargv, envp);

    cout << "p" << p3.p.pid << " command line: ";
    parse(command, pargc, pargv);
    for (int i = 0; i < pargc; i++) {
        cout << "argv[" << i << "] = ";
        for (int j = 0; pargv[i][j] != '\0'; j++) {
            cout << pargv[i][j];
        }
        cout << endl;
    }
    p3.initialize(&main_mem, pargc, pargv, envp);
}

void dual_mips_shared_l2::evaluate() {
    p0.evaluate();
    p1.evaluate();
    p2.evaluate();
    p3.evaluate();
```

```
}

void dual_mips_shared_l2::evaluate_end() {
    p0.evaluate_end();
    p1.evaluate_end();
    p2.evaluate_end();
    p3.evaluate_end();

    //performance measures
    if (bus1.con_chan.read() != 0)
        bus1_busy_count += 1;

}

void dual_mips_shared_l2::dump(ostream &out) const {
    p0.dump(out);
    p0_req12i.dump(out);
    p0_grant12i.dump(out);
    p0_req12d.dump(out);
    p0_grant12d.dump(out);
    p1.dump(out);
    p1_req12i.dump(out);
    p1_grant12i.dump(out);
    p1_req12d.dump(out);
    p1_grant12d.dump(out);
    p2.dump(out);
```

```
    p2_req12i.dump(out);
    p2_grant12i.dump(out);
    p2_req12d.dump(out);
    p2_grant12d.dump(out);
    p3.dump(out);
    p3_req12i.dump(out);
    p3_grant12i.dump(out);
    p3_req12d.dump(out);
    p3_grant12d.dump(out);

    arbiter.dump(out);
    req12.dump(out);
    grant12.dump(out);
    bus1.dump(out);
    cache2.dump(out);
    req23.dump(out);
    grant23.dump(out);
    bus2.dump(out);
    main_mem.dump(out);
    mem_manager.dump(out);
  }
```

Replace your dual_mips_shared_l2.h with the following:

```
#ifndef DUAL_MIPS_SHARED_L2_H
#define DUAL_MIPS_SHARED_L2_H

#include "cache.h"
#include "main_mem.h"
#include "arbiter.h"
#include "mips_l1.h"


struct dual_mips_shared_l2: public sc_module {
   //ports
   ab_clk_in clk;

   //local channels
   ab_signal<bool> p0_req12i;
   ab_signal<bool> p0_grant12i;
   ab_signal<bool> p0_req12d;
   ab_signal<bool> p0_grant12d;
   ab_signal<bool> p1_req12i;
   ab_signal<bool> p1_grant12i;
   ab_signal<bool> p1_req12d;
   ab_signal<bool> p1_grant12d;

   ab_signal<bool> p2_req12i;
   ab_signal<bool> p2_grant12i;
   ab_signal<bool> p2_req12d;
```

```
ab_signal<bool> p2_grant12d;
ab_signal<bool> p3_req12i;
ab_signal<bool> p3_grant12i;
ab_signal<bool> p3_req12d;
ab_signal<bool> p3_grant12d;

ab_signal<bool> req12;
ab_signal<bool> grant12;
ab_signal<bool> req23;
ab_signal<bool> grant23;
ab_memory_bus bus1;
ab_memory_bus bus2;

//sub-module instances
ab_host_mem_manager mem_manager;
mips_l1 p0;
mips_l1 p1;
mips_l1 p2;
mips_l1 p3;
ab_arbiter<8> arbiter;
ab_cache cache2;
ab_cond_count cache2_access_count;
ab_cond_count cache2_miss_count;
ab_main_mem main_mem;
ab_cond_count main_mem_access_count;
```

```
    //constructor
    dual_mips_shared_l2();
    explicit dual_mips_shared_l2(sc_module_name name_);

    void initialize(char **envp);
    void evaluate();
    void evaluate_end();

    void dump(ostream &out) const;

    mips_l1 **pl1ptr;
    long bus1_busy_count;
};

#endif
```

Replace your testbench.cpp with the following:

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <string>
using std::string;
#include "testbench.h"
#include "trace.h"
```

```
sc_time_unit ab_time_unit = SC_NS;

ab_testbench::ab_testbench(sc_module_name name_, int argc_, char **argv_,
    char **envp_)
    :
    sc_module(name_),
    //local channel initialization

    //sub-module instance initialization
    clkgen("clkgen", 2.0, ab_time_unit),
    top("top"),

    argc(argc_),
    argv(argv_),
    envp(envp_),
    print_cycle(0),
    display_cycle(0),
    break_cycle(0),
    interactive(0)
{
    //sub-module port connection
    top.clk(clkgen.clk);

    //processes
    SC_METHOD(process);
        sensitive << clkgen.clk;
```

```
    SC_METHOD(process_end);
        sensitive << clkgen.clk_end;
        dont_initialize();
}

void ab_testbench::start_of_simulation() {
    //command line options
    while ((argc > 1) && (argv[1][0] == '-')) {
        switch( argv[1][1] ) {
          case 'd':
            interactive = 1;
            argc -= 1;
            argv += 1;
            break;
              case 'h':
                cout << "usage: dual_mips_shared_l2 [option] ... [option] " << endl;
                cout << "options: " << endl;
                cout << "  -d (interactive debug)" << endl;
                cout << "  -V (version)" << endl;
                cout << "  -p n (print stats every n cycles)" << endl;
            SC_REPORT_ERROR("abakus", name());
              case 'p':
                print_cycle = 0;
                for (int i = 0; argv[2][i] != '\0'; i++) {
                        print_cycle *= 10;
                        print_cycle += (long) argv[2][i] - (long) '0';
```

```
            }
            cout << "print_cycle = " << print_cycle << endl;
        argc -= 2;
        argv += 2;
        break;
          case 'V':
            cout << argv[0] << " version 0.2.0" << endl;
        argc -= 1;
        argv += 1;
        break;
       default:
        cout << "unrecognized option " << argv[1] << endl;
        SC_REPORT_ERROR("abakus", name());
      }
   }

   //processor initialization
   argc -=1;
   argv +=1;
   top.initialize(envp);

   if (interactive) debug_interaction();
}

void ab_testbench::process() {
   //evaluate submodule proceses first
```

```
    top.evaluate();
}


void ab_testbench::process_end() {
    //evaluate submodule proceses first
    top.evaluate_end();


    if (ab_debug) {
            dumpfile << "\n\ncycle number " << ab_clk_count
            << " at " << sc_time_stamp() << endl;
            top.p0.dump(dumpfile);
            top.p1.dump(dumpfile);
            top.p2.dump(dumpfile);
        top.p3.dump(dumpfile);
        top.arbiter.dump(dumpfile);
        top.cache2.dump(dumpfile);
        top.bus2.dump(dumpfile);
        top.main_mem.dump(dumpfile);
        dumpfile << "+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+\n\n";
    }


    if (interactive) {
        debug_monitor();
    }
    if ((print_cycle != 0) && (ab_clk_count % print_cycle == 0)) print_stats();
}
```

```
void ab_testbench::dump(ostream &out) const {
   out << endl << name() << endl;
   clkgen.dump(out);
   top.dump(out);
}

                struct timeval begin_u, end_u, begin_s, end_s;
                struct rusage usg;
                float user_time, sys_time;

int sc_main(int argc, char **argv) {

   cout << "in sc_main" << endl;

   sc_set_time_resolution(1.0, ab_time_unit);
   //make sc_time objects after setting time resolution
   ab_half_cycle = new sc_time(1.0, ab_time_unit);

   cout.unsetf(ios::dec);
   cout.setf(ios::hex);
   cout.width(2*sizeof(data_t) );
   cout.fill('0');

   //fake environment
   char *p = 0;
```

```
    char* *envp = &p;

    //start elaboration
    ab_testbench test("ab_testbench", argc, argv, envp);

    //set up elapsed time measurement
                getrusage(RUSAGE_SELF, &usg);
                begin_u = usg.ru_utime;
                begin_s = usg.ru_stime;

    cout << "calling sc_start" << endl;
    //sc_start(200000.0, ab_time_unit);
    sc_start();
    cout << "finished sc_start" << endl;
    test.print_stats();
    return(0);
}

void ab_testbench::print_stats() {
    //finish elapsed time measurement
                getrusage(RUSAGE_SELF, &usg);
                end_u = usg.ru_utime;
                end_s = usg.ru_stime;
                user_time = (end_u.tv_sec+end_u.tv_usec/1000000.0)-
                      (begin_u.tv_sec+begin_u.tv_usec/1000000.0);
                sys_time = (end_s.tv_sec+end_s.tv_usec/1000000.0)-
```

```
                    (begin_s.tv_sec+begin_s.tv_usec/1000000.0);
    cout.unsetf(ios::hex);
    cout.setf(ios::dec);
    cout.fill(' ');
    cout << endl;
    cout << "clock cycles: " << ab_clk_count << endl;
    top.p0.perf_sum();
    top.p1.perf_sum();
    top.p2.perf_sum();
    top.p3.perf_sum();
    cout << endl;
    cout << "bus 1 busy cycles: " << top.bus1_busy_count
         << " busy rate: " << (float) top.bus1_busy_count
                 / (float) ab_clk_count << endl;
    cout << "l2-cache accesses: " << top.cache2_access_count.count
<< " misses: " << top.cache2_miss_count.count
<< " miss rate " << (float) top.cache2_miss_count.count
           / (float) top.cache2_access_count.count << endl;
    cout << "main mem accesses: " << top.main_mem_access_count.count
         << endl;
    cout << endl;

    cout << "Total user time:   " << user_time << endl;
    cout << "Total system time: " << sys_time << endl;
    cout << "Simulation speed (cyc/sec): "
         << ab_clk_count/(user_time + sys_time) << endl;
```

```
    cout << "Simulation host mem swaps: " << top.mem_manager.swap_count
         << endl;
    float swaps_per_access = (float) top.mem_manager.swap_count
         / (float) (top.p0.if_nop_count
            + top.p0.if_icount
            + top.p0.dcache_access_count.count
            + top.p1.if_nop_count
            + top.p1.if_icount
            + top.p1.dcache_access_count.count
            + top.p2.if_nop_count
            + top.p2.if_icount
            + top.p2.dcache_access_count.count
            + top.p3.if_nop_count
            + top.p3.if_icount
            + top.p3.dcache_access_count.count
            + top.cache2_access_count.count
            + top.main_mem_access_count.count);
    cout << "Swaps per memory access: " << swaps_per_access << endl;

    cout.unsetf(ios::dec);
    cout.setf(ios::hex);
    cout.width(2*sizeof(data_t) );
    cout.fill('0');
}
```

VITA

Julius Jonggara Raya Hot Marisi Marpaung

Candidate for the Degree of

Doctor of Philosophy/Education

Thesis: PERFORMANCE LIMITATIONS FOR MULTICORE PROCESSORS

Major Field: Electrical and Computer Engineering

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2012.

Completed the requirements for the Master of Science in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2006.

Completed the requirements for the Bachelor of Science in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2003.

Experience:

Teaching Assistant for ENGR 1342, ECEN 4213, ECEN 3213, ECEN 4243, and ECEN 3233
Research Assistant for Dr. Louis Johnson
Lecturer for ECEN 3233 Digital Logic Design

Professional Memberships:

Eta Kappa Nu
National Society of Collegiate Scholars
Golden Key

Name: Julius Jonggara Raya Hot Marisi Marpaung Date of Degree: May, 2012

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: PERFORMANCE EVALUATIONS FOR MULTICORE PROCESSORS

Pages in Study: 106 Candidate for the Degree of Doctor of Philosophy

Major Field: Electrical and Computer Engineering

Scope and Method of Study: To use and improve a new simulation tool that emulates and studies different cache hierarchies and configurations to evaluate the performance of any chosen processor and cache configurations.

Findings and Conclusions: Sharing a L2 cache with more than eight processors may reduce performance. Using a shared L3 cache or hierarchical architecture may result in a better performance. The major factor that contributes to the loss of performance is the bus contention. Increasing the size of shared cache does not have a significant impact on performance.

ADVISER'S APPROVAL: Dr. Louis Johnson